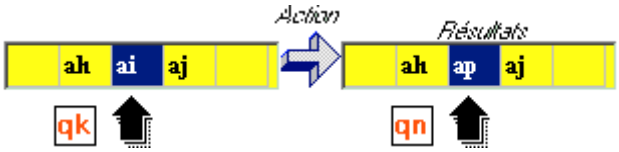
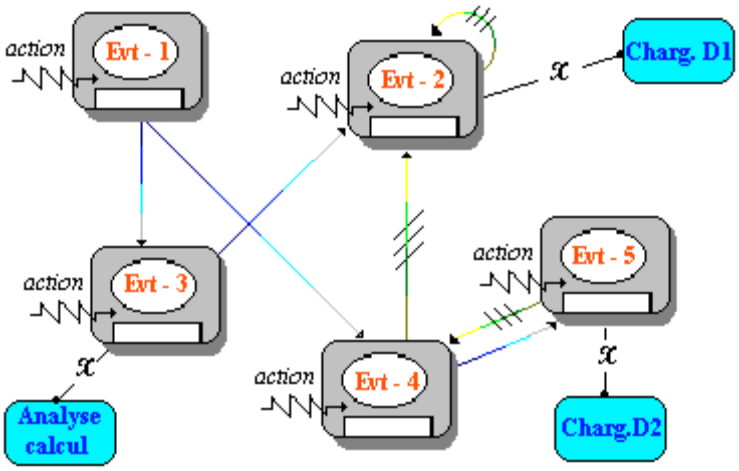
Les bases de l'informatique



et de
la programmation



RM di Scala *– révision du 05 Septembre 2004*

914 pages

SOMMAIRE

Introduction 4

Chapitre 1.La machine

- 1.1.Ordinateur et évolution 6
- 1.2.Les circuits logiques 14
- 1.3.Codage et numération 28
- 1.4.Formalisation de la notion d'ordinateur 39
- 1.5.Architecture de l'ordinateur 50
- 1.6.Système d'exploitation 62
- 1.7.Les réseaux 69
- Exercices avec solutions 83

Chapitre 2.Programmer avec un langage

- 2.1.Les langages 84
- 2.2.Relations binaires 90
- 2.3.Théorie des langages 96
- 2.4.Les bases du langage Delphi 112
- Exercices avec solutions 147

Chapitre 3.Développer du logiciel avec méthode

- 3.1.Développement méthodique du logiciel 147
- .Machines abstraites : exemple 181
- 3.2.Modularité 191
- 3.3.Complexité, tri, recherche 200
- tri à bulle 208

- tri par selection 214
- tri par insertion 222
- tri rapide 228
- tri par tas 238
- recherche en table 258
- Exercices avec solutions 264

Chapitre 4. Structures de données

- 4.1.spécifications abstraites de données 278
- 4.2 types abstraits TAD et implantation 292
- exercice TAD et solution d'implantation 304
- 4.3 structures d'arbres binaires 319
- Exercices avec solutions 337

Chapitre 5. Programmation objet et événementielle

- 5.1.Introduction à la programmation orientée objet 348
- 5.2.Programmez objet avec Delphi 363
- 5.3.Polymorphisme avec Delpbi 390
- 5.4.Programmation événementielle et visuelle 424
- 5.5.Les événements avec Delphi 443
- 5.6.Programmation défensive 470
- Exercices avec solutions 486

Chapitre 6. Programmez avec des grammaires

- 6.1.Programmation avec des grammaires 509
- 6.2.Automates et grammaires de type 3 532
- 6.3.projet de classe mini-interpréteur 551
- 6.4.projet d 'indentateur de code 571
- Exercices avec solutions 595

Chapitre 7. Communication homme-machine

- 7.1.Les interfaces de communication logiciel/utilisateur 611
- 7.2. Grammaire pour analyser des phrases 618
- 7.3. Interface et pilotage en mini-français 638
- 7.4. Projet d'IHM : enquête fumeurs 658
- 7.5. Utilisation des bases de données 670
- Exercices avec solutions 703

Chapitre 8. Les composants sont des logiciels réutilisables

- 8.1.Construction de composants avec Delphi 718
- 8.2. Les messages windows avec Delphi 759
- 8.3. Création d'un événement associé à un message 780
- 8.4. ActiveX avec la technologie COM 787
- Exercices avec solutions 805

Introduction

Issu d'un cours de programmation à l'université de Tours en premier cycle scientifique, en DESS, Master Sciences et technologie compétence complémentaire informatique et en Diplôme Universitaire (DU) compétence complémentaire informatique pour les NTIC (réservés à des non-informaticiens), cet ouvrage est une synthèse (non exhaustive)sur les minima à connaître sur le sujet.

Il permettra au lecteur d'aborder la programmation objet et l'écriture d'interfaces objets événementielles sous Windows en particulier.

Ce document sera utile à un public étudiant (IUT info, BTS info, IUP informatique et scientifique, DEUG sciences, licence pro informatique, DESS et DU compétence complémentaire en informatique) et de toute personne désireuse de se former par elle-même (niveau prérequis Bac scientifique).

Le premier chapitre rassemble les concepts essentiels sur la notion d'ordinateur, de codage, de programme et d'instruction au niveau machine.

Le second chapitre introduit le concept de langage de programmation et de grammaire de chomsky, le langage pascal sert d'exemple.

Le chapitre trois montre comment utiliser des grammaires pour programmer en mode génération ou en mode analyse.

Le chapitre quatre forme le noyau dur d'une approche méthodique pour développer du logiciel, les thèmes abordés sont : algorithme, complexité, programmation descendante, machines abstraites, modularité, types abstraits. Ce chapitre fournit aussi des outils de tris sur des tableaux et des algorithmes de traitement d'arbres binaires.

Le chapitre cinq contient les éléments fondamentaux de la programmation orientée objet, du polymorphisme d'objet, du polymorphisme de méthode, de la programmation événementielle et visuelle, de la programmation défensive et de construction d'interface homme-machine. Le langage Delphi sert de support à l'implantation pratique de ces notions essentielles.

Le chapitre six montre comment utiliser la programmation par composant avec Delphi, puis aborde le traitement des messages dans Windows et comment programmer des applications utilisant les messages système pour communiquer.

Chapitre 1 : La machine

1.1.Ordinateur et évolution

- les 3 grandes lignes de pensées
- les générations d'ordinateurs
- l'ordinateur
- information-informatique

1.2.Les circuits logiques

- logique élémentaire pour l'informatique
- algèbre de Boole
- circuits booléens

1.3.Codage numération

- codage de l'information
- numération

1.4.Formalisation de la notion d'ordinateur

- machine de Türing théorique
- machine de Türing physique

1.5.Architecture de l'ordinateur

- les principaux constituants
- mémoires, mémoire centrale
- une petite machine pédagogique

1.6.Système d'exploitation

- notion de système d'exploitation
- systèmes d'exploitation des micro-ordinateurs

1.7.Les réseaux

- les réseaux d'ordinateurs
- liaisons entre réseaux

1.1 Ordinateur et évolution

Plan du chapitre:

1. Les 3 grandes lignes de pensée

1.1 Les machines à calculer
1.2 Les automates
1.3 Les machines programmables

2. Les générations de matériels

2.1 Première génération 1945-1954
2.2 Deuxième génération 1955-1965
2.3 Troisième génération 1966-1973
2.4 Quatrième génération à partir de 1974

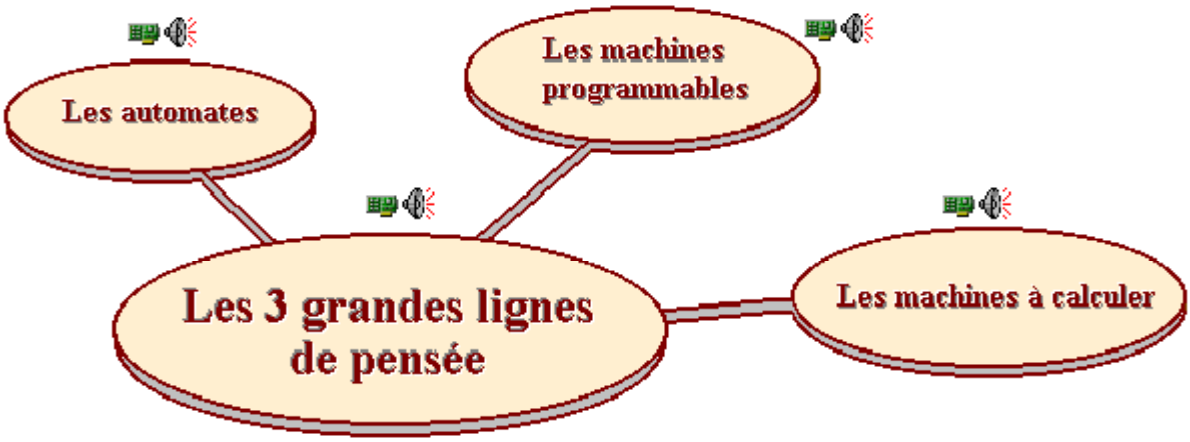
3. L'ordinateur

3.1 Utilité de l'ordinateur
3.2 Composition minimale d'un ordinateur
3.3 Autour de l'ordinateur : les périphériques
3.4 Pour relier tout le monde

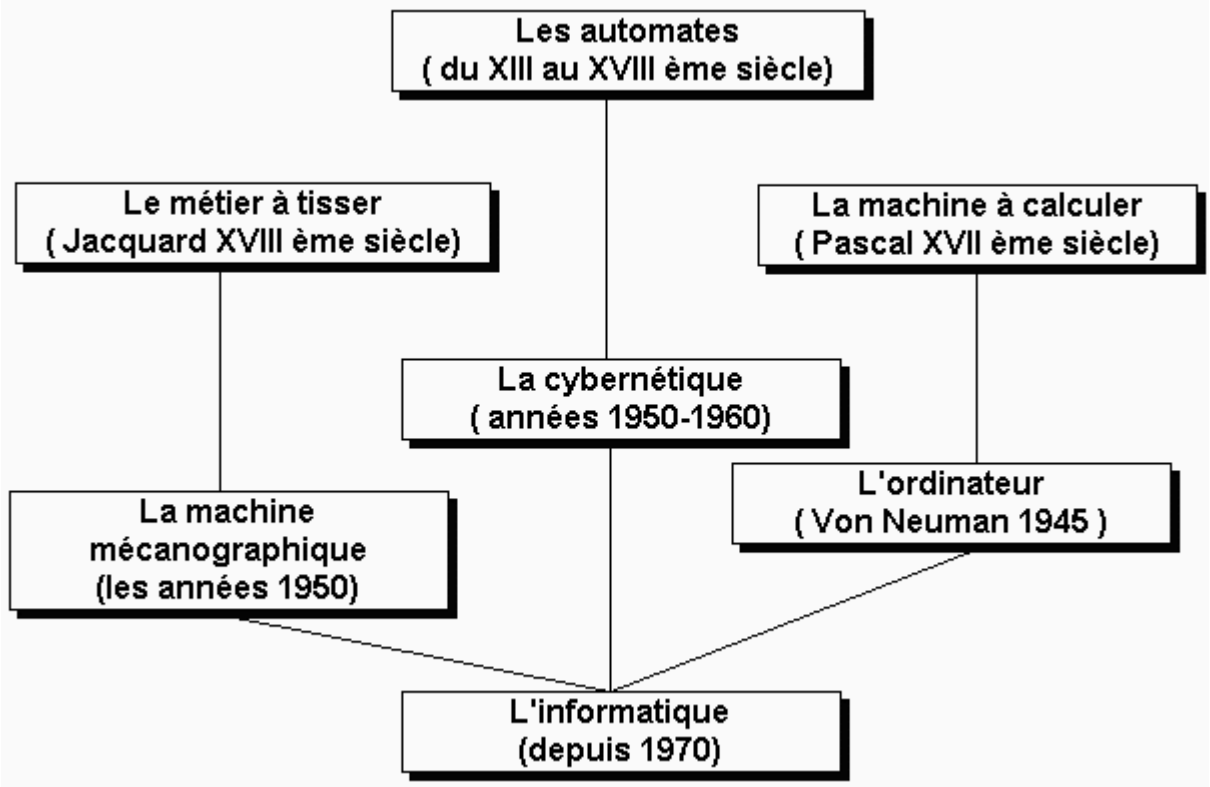
4. Information - Informatique

4.1 Les définitions
4.2 Critère algorithmique élémentaire

1. Les 3 grandes lignes de pensée



L'histoire de l'informatique débute par l'invention de machines (la fonction crée l'organe) qui au départ correspondent à des lignes de pensée différentes. L'informatique résultera de la fusion des savoirs acquis dans ces domaines. Elle n'est pas une synthèse de plusieurs disciplines, mais plutôt une discipline entièrement nouvelle puisant ses racines dans le passé. Seul l'effort permanent du génie créatif humain l'a rendue accessible au grand public de nos jours.



1.1 Les machines à calculer

La Pascaline de Pascal, 17ème siècle. Pascal invente la Pascaline, première machine à calculer (addition et soustraction seulement), pour les calculs de son père.

La machine multiplicatrice de Leibniz, 17ème siècle. Leibniz améliore la machine de Pascal pour avoir les quatre opérations de base (+,-,*,/).

1.2 Les automates

Les automates, les horloges astronomiques, les machines militaires dès le 12ème siècle.

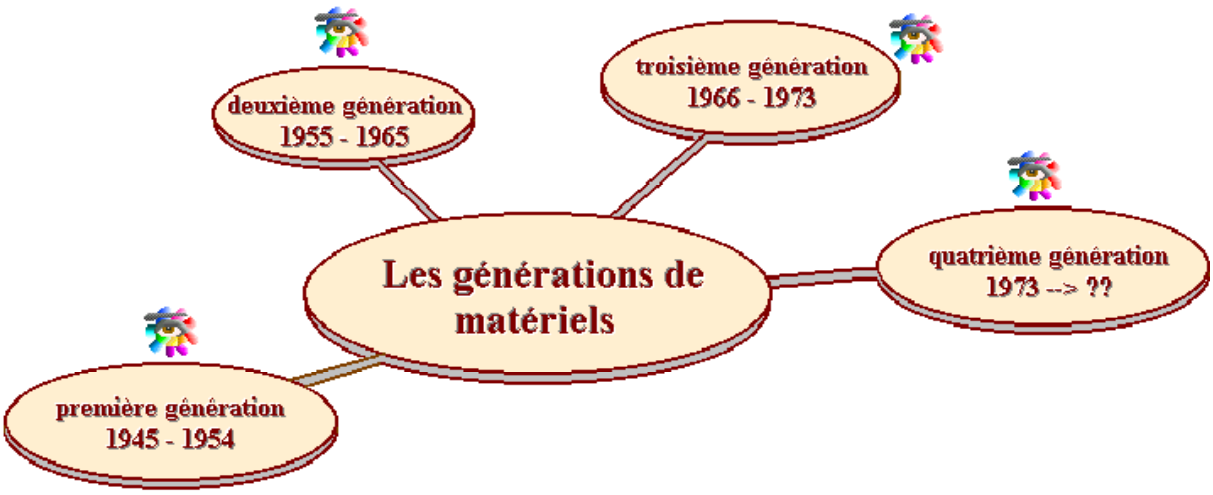
1.3 Les machines programmables

Le métier à tisser de **Jacquard**, 1752-1834

Début de commercialisation des machines mécaniques scientifiques (usage militaire en général).

Babage invente la première machine analytique programmable.

2. Les générations de matériels



On admet généralement que l'ère de l'informatique qui couvre peu de décennies se divise en plusieurs générations essentiellement marquées par des avancées technologiques

2.1 Première génération 1945 - 1954

Informatique scientifique et militaire.

Il faut résoudre les problèmes des calculs répétitifs.

Création de langages avec succès et échecs dans le but de résoudre les problèmes précédents.
Technologie lourde (Tube et tore de ferrite), qui pose des problèmes de place et de consommation électrique.

Les très grandes nations seules possèdent l'outil informatique.

2.2 Deuxième génération 1955-1965

Naissance de l'informatique de gestion.

Nouvelle technologie basée sur le transistor et le circuit imprimé. Le langage **Fortran** règne en maître incontesté. Le langage de programmation **Cobol** orienté gestion, devient un concurrent de **Fortran**.

Les nations riches et les très grandes entreprises accèdent à l'outil informatique.

2.3 Troisième génération 1966-1973

Naissance du circuit intégré.

Nouvelle technologie basée sur le **transistor** et le circuit intégré.

Les ordinateurs occupent moins de volume, consomment moins d'électricité et sont plus rapides. Les ordinateurs sont utilisés le plus souvent pour des applications de gestion.

Les PME et PMI de tous les pays peuvent se procurer des matériels informatiques.

2.4 Quatrième génération à partir de 1974

Naissance de la micro-informatique

La création des microprocesseurs permet la naissance de la micro-informatique(le micro-ordinateur Micral de R2E est inventé par un français **François Gernelle** en 1973). Steve Jobs (Apple) invente un nouveau concept vers la fin des années 70 en recopiant et en commercialisant les idées de Xerox parc à travers le MacIntosh et son interface graphique.

Un individu peut actuellement acheter son micro-ordinateur dans un supermarché.

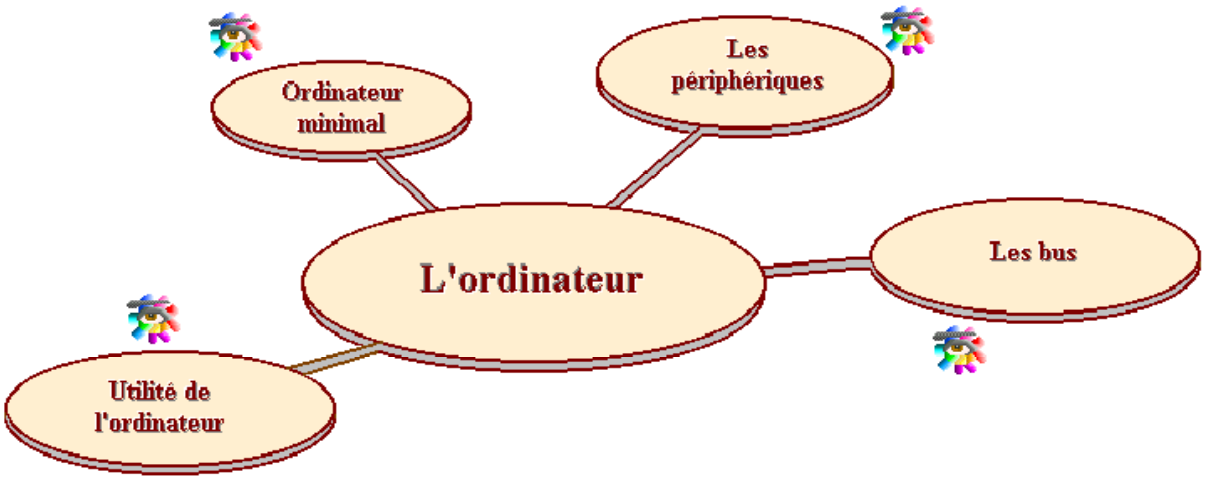
Nous observons un phénomène fondamental :

La démocratisation d'une science à travers un outil. L'informatique qui à ses débuts était une affaire de spécialistes, est aujourd'hui devenue l'affaire de tous; d'où l'importance d'une solide formation de tous aux différentes techniques utilisées par la science informatique, car la banalisation d'un outil ou d'une science a son revers : l'assoupissement de l'attention envers les inconvénients inhérents à tout progrès technique.

Tableau synoptique des générations d'ordinateurs :

	1ère Génér. 45 - 54	2ème Génér. 55 - 65	3ème Génér. 66 - 73	4ème Génér. 74 ---> ?
composants	tubes radios	transistor	circuit intégré	VLSI
mémoires	tubes/ tores de ferrite	tores de ferrite	circuit intégré	VLSI
temps de traitement	10^{-2} s	10^{-3} s	10^{-6} s	10^{-9} s
système d'exploitation	rudimentaire	monoprogram -mation	multiprogram -mation	temps-partagé
rendement %	3 %	10%	30 %	60 %

3. L'ordinateur



3.1 Utilité de l'ordinateur

Un ordinateur est une machine à traiter de l'information.

L'information est fournie sous forme de données traitées par des programmes (exécutés par des ordinateurs).

3.2 Composition minimale d'un ordinateur : **le cœur**

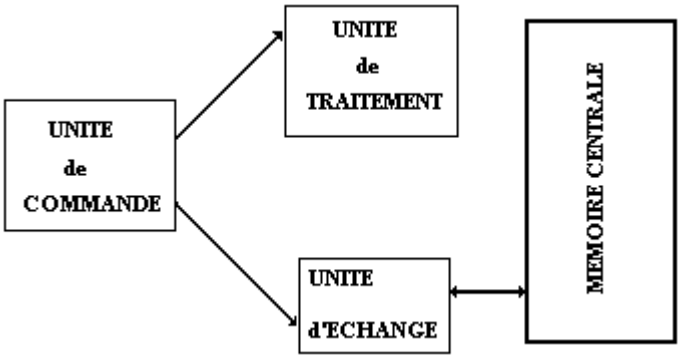
Une mémoire Centrale .

Une unité de traitement avec son UAL (unité de calcul).

Une unité de commande ou contrôle.

Une ou plusieurs unités d'échanges.

Schéma simplifié du cœur de l'ordinateur



3.3 Autour de l'ordinateur : les périphériques

Les périphériques sont chargés d'effectuer des tâches d'entrées et/ou de sortie de l'information. En voici quelques uns.

Périphériques d'entrée
Clavier, souris, crayon optique, écran tactile, stylo code barre, carte son, scanner, caméra, etc.

Périphériques de sortie
Ecran, imprimante, table traçante, carte son, télécopie, modem etc.

Périphériques d'entrée sortie
Mémoire auxiliaire (sert à stocker les données et les programmes):

1. Stockage de masse sur disque dur ou disquette.
2. Bande magnétique sur dérouleur (ancien) ou sur streamer.
3. Mémoire clef USB
4. CD-Rom, DVD, disque magnéto-électrique etc…

3.4 Pour relier tout le monde : Les Bus

Les Bus représentent dans l'ordinateur le système de communication entre ses divers constituants. Ils sont au nombre de trois :

le **Bus d'adresses**, (la notion d'adresse est présentée plus loin)

le **Bus de données**,

le **Bus de contrôle**.

4. Information - informatique

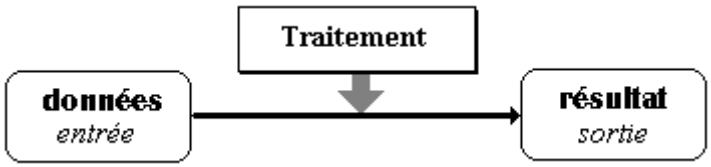
4.1 Les définitions

L'**information** est le support formel d'un élément de connaissance humaine susceptible d'être représentée à l'aide de conventions (codages) afin d'être conservée, traitée ou communiquée.

L'**informatique** est la science du traitement de l'information dans les domaines scientifiques, techniques, économiques et sociaux.

Une **donnée** est la représentation d'une information sous une forme conventionnelle (codée) destinée à faciliter son traitement.

schéma simplifié du traitement de l'information



4.2 Critère algorithmique élémentaire

Une application courante est justiciable d'un traitement informatique si :

Il est possible de définir et de décrire parfaitement les données d'entrée et les résultats de sortie.
Il est possible de décomposer le passage de ces données vers ces résultats en une suite d'opérations élémentaires dont chacune peut être exécutée par une machine.

Nous pouvons considérer ce critère comme une définition provisoire d'un algorithme.

Actuellement l'informatique intervient dans tous les secteurs d'activité de la vie quotidienne :

démontrer un théorème *(mathématique)* faire jouer aux échecs *(intelligence artificielle)* dépouiller un sondage *(économie)*
gérer un robot industriel *(atelier)* facturation de produits *(entreprise)* traduire un texte *(linguistique)* imagerie médicale *(médecine)* formation à distance *(éducation)* Internet *(grand public)*...etc

1.2 Les circuits logiques

Plan du chapitre:

1. Logique élémentaire pour l'informatique

1.1 Calcul propositionnel naïf
1.2 Propriétés des connecteurs logiques
1.3 Règles de déduction

2. Algèbre de Boole

2.1 Axiomatique pratique
2.2 Exemples d'algèbre de Boole
2.3 Notation des électroniciens

3.Circuits booléens ou logiques

3.1 Principaux circuits
3.2 Fonction logique associée à un circuit
3.3 Circuit logique associé à une fonction
3.4 Additionneur dans l'UAL

1. Logique élémentaire pour l'informatique

1.1 Calcul propositionnel naïf

Construire des programmes est une activité scientifique fondée sur le raisonnement logique. Un peu de logique simple va nous aider à disposer d'outils pratiques mais rigoureux pour construire des programmes les plus justes possibles. Si la programmation est un art, c'est un art rigoureux et logique. La rigueur est d'autant plus nécessaire que les systèmes informatiques manquent totalement de sens artistique.

Une proposition est une propriété ou un énoncé qui peut avoir une valeur de vérité vraie (notée **V**) ou fausse (notée **F**).

" 2 est un nombre impair " est une proposition dont la valeur de vérité est **F**.

Par abus de langage nous noterons avec **le même symbole une proposition et sa valeur de vérité**, car seule la valeur de vérité d'une proposition nous intéresse ici.

Soit l'ensemble P = { **V , F** } des valeurs des propositions.

On le munit de trois opérateurs appelés connecteurs logiques : $\neg$, $\wedge$, $\vee$**.**

$\wedge$: P x P$\rightarrow$P (se lit " **et** ")

$\vee$: P x P$\rightarrow$P (se lit " **ou** ")

$\neg$: P $\rightarrow$P (se lit " **non** ")

Ces connecteurs sont définis en extension par leur tables de vérité :

p	q	$\neg$ p	p $\wedge$ q	p $\vee$ q
V	**V**	**F**	**V**	**V**
V	**F**	**F**	**F**	**V**
F	**V**	**V**	**F**	**V**
F	**F**	**V**	**F**	**F**

1.2 Propriétés des connecteurs logiques

- Nous noterons p = q , le fait la proposition p et la proposition q ont la même valeur de vérité.
- Le lecteur pourra démontrer à l'aide des tables de vérité par exemple, que $\vee$ et $\wedge$ possèdent les propriétés suivantes :
  - $p \vee q = q \vee p$
  - $p \wedge q = q \wedge p$
  - $p \vee (q \vee r) = (p \vee q) \vee r$
  - $p \wedge (q \wedge r) = (p \wedge q) \wedge r$
  - $p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$
  - $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$
  - $p \vee p = p$
  - $p \wedge p = p$
  - $\neg\neg p = p$
  - $\neg (p \vee q) = \neg p \wedge \neg q$
  - $\neg (p \wedge q) = \neg p \vee \neg q$
- Nous notons $p \Rightarrow q$, la proposition : $\neg p \vee q$ (l'implication).

Table de vérité du connecteur $\Rightarrow$:

p	q	$p \Rightarrow q$
V	**V**	**V**
V	**F**	**F**
F	**V**	**V**
F	**F**	**V**

- Il est aussi possible de prouver des " égalités " de propositions en utilisant des combinaisons de résultats précédents.

Exemple : Montrons que : $p \Rightarrow q = \neg q \Rightarrow \neg p$ (implication contrapposée), par définition et utilisation évidente des propriétés :

$p \Rightarrow q = \neg p \vee q = q \vee \neg p = \neg(\neg q) \vee \neg p = \neg q \Rightarrow \neg p$

1.3 Règles de déduction

Assertion :
c'est une proposition construite à l'aide des connecteurs logiques ($\neg$, $\wedge$, $\vee$, en particulier) dont la valeur de vérité est toujours V (vraie).

Les règles de déduction permettent de faire du calcul sur les *assertions*. Nous abordons ici le raisonnement rationnel sous son aspect automatisable, en donnant des règles d'inférences extraites du modèle du raisonnement logique du logicien **Gentzen**. Elles peuvent être une aide très appréciable lors de la construction et la spécification d'un programme.

Les règles de déduction sont séparées en deux membres. Le premier contient les prémisses ou hypothèses de la règle, le deuxième membre est constitué par une conclusion unique. Les deux membres sont séparés par un trait horizontal. Gentzen classe les règles de déduction en deux catégories : les règles d'introduction car il y a utilisation d'un nouveau connecteur, et les règles d'éliminations qui permettent de diminuer d'un connecteur une proposition.

Syntaxe d'une telle règle : $\dfrac{p1, p2, \ldots, pn}{q}$

Quelques règles de déductions pratiques :

Règle d'introduction du $\wedge$: $\dfrac{p\ ,\ q}{p \wedge q}$

Règle d'introduction du $\vee$: $\dfrac{p\ ,\ q}{p \vee q}$

Règle d'introduction du $\Rightarrow$: $\dfrac{p\ ,\ q}{p \Rightarrow q}$

Règles d'élimination du $\wedge$: $\dfrac{p \wedge q}{q}$, $\dfrac{p \wedge q}{p}$

Règle du modus ponens : $\dfrac{p \ , \ p \Rightarrow q}{q}$

Règle du modus tollens : $\dfrac{\neg q \ , \ p \Rightarrow q}{\neg p}$

Le système de **Gentzen** contient d'autres règles sur le **ou** et sur le **non**. Enfin il est possible de construire d'autres règles de déduction à partir de celles-ci et des propriétés des connecteurs logiques. Ces règles permettent de prouver la valeur de vérité d'une proposition. Dans les cas pratiques l'essentiel des raisonnements revient à des démonstrations de véracité d'implication.

La démarche est la suivante : pour prouver qu'une implication est vraie, il suffit de supposer que le membre gauche de l'implication est vrai et de montrer que sous cette hypothèse le membre de droite de l'implication est vrai.

Exemple **:**
soit à montrer que la règle de déduction $\mathbf{R_0}$ suivante est exacte :

$\mathbf{R_0}$ **:** $\dfrac{p \Rightarrow q \ , \ q \Rightarrow r}{p \Rightarrow r}$ *(transitivité de $\Rightarrow$)*

*Hypothèse : **p** est vrai*
nous savons que : $\mathbf{p} \Rightarrow \mathbf{q}$ est vrai et que $\mathbf{q} \Rightarrow \mathbf{r}$ est vrai

- En appliquant le modus ponens : $\dfrac{p \ , \ p \Rightarrow q}{q}$ nous déduisons que : **q** est vrai.
- En appliquant le modus ponens à ($\mathbf{q}$ **,** $\mathbf{q} \Rightarrow \mathbf{r}$) nous déduisons que : **r** est vrai.
- Comme p est vrai (par hypothèse) on applique la règle d'introduction de $\Rightarrow$ sur (**p , r**) nous déduisons que : $\mathbf{p} \Rightarrow \mathbf{r}$ est vrai **(cqfd).**

Nous avons prouvé que $\mathbf{R_0}$ est exacte. Ainsi nous avons construit une nouvelle règle de déduction qui pourra nous servir dans nos raisonnements.

Nous venons d'exhiber un des outils permettant la construction raisonnée de programmes. La logique interne des ordinateurs est encore actuellement plus faible puisque basée sur la logique booléenne, en attendant que les machines de 5ème génération basées sur la logique du premier ordre (logique des prédicats, supérieure à la logique propositionnelle) soient définitivement opérationnelles.

2. Algèbre de Boole

2.1 Axiomatique pratique

Du nom du mathématicien anglais **G.Boole** qui l'inventa. Nous choisissons une axiomatique compacte, l'axiomatique algébrique :

On appelle algèbre de Boole tout ensemble E muni de :

deux lois de compositions internes notées par exemple : $\bullet$ et $\oplus$,

- une application involutive f (f^2 = Id) de E dans lui-même,notée $f : a \longrightarrow \bar{a}$
- chacune des deux lois $\bullet$, $\oplus$, est associative et commutative,
- chacune des deux lois $\bullet$, $\oplus$, est distributive par rapport à l'autre,
- la loi $\bullet$ admet un élément neutre unique noté e_1,

  $x \in E,\ x \bullet e_1 = x$

- *la loi* $\oplus$ admet un élément neutre noté e_0,

  $x \in E,\ x \oplus e_0 = x$

- *tout élément de E est idempotent pour chacune des deux lois :*

  $x \in E,\ x \bullet x = x$ et $x \oplus x = x$

- *axiomes de complémentarité* :

  $\forall x \in E, x \bullet \bar{x} = e_0$
  $\forall x \in E, x \oplus \bar{x} = e_1$

- *lois de Morgan* :

  $(x,y) \in E^2,\ \overline{x \bullet y} = \bar{x} \oplus \bar{y}$

  $(x,y) \in E^2,\ \overline{x \oplus y} = \bar{x} \bullet \bar{y}$

2.2 Exemples d'algèbre de Boole

a) L'ensemble P(E) des parties d'un ensemble E, muni des opérateurs intersection $\cap$,union $\cup$, et l'application involutive complémentaire dans E $\rightarrow C_E$, (si E $\neq \varnothing$), si E est fini et possède n éléments, P(E) est de cardinal 2^n.

Il suffit de vérifier les axiomes précédents en substituant les lois du nouvel ensemble E aux lois $\bullet$, $\oplus$, et $\bar{x}$. Il est montré en mathématiques que toutes les algèbres de Boole finies sont isomorphes à un ensemble (P(E), $\cap$, $\cup$, C_E) : elles ont donc un cardinal de la forme 2^n.

b) L'ensemble des propositions (en fait l'ensemble de leurs valeurs {**V**, **F**}) muni des connecteurs logiques $\neg$ (l'application involutive) , $\wedge$, $\vee$ **, est une algèbre de Boole minimale à deux éléments.**

2.3 Notation des électroniciens

L'algèbre des circuits électriques est **une algèbre de Boole minimale à deux éléments** :

L'ensemble E = {**0**,**1**} muni des lois " $\bullet$ " et " + " et de l'application complémentaire $f : a \longrightarrow \bar{a}$.

Formules pratiques et utiles (résultant de l'axiomatique) :

a + 1 = 1	a.1 = a
a + 0 = a	a.0 = 0
a + a = a	a.a = a
$a + \bar{a} = 1$	$a \,.\, \bar{a} = 0$
$\overline{a + b} = \bar{a} \,.\, \bar{b}$	$\overline{a \,.\, b} = \bar{a} + \bar{b}$
$\bar{0} = 1$	$\bar{1} = 0$

Formule d'absorbtion :

a+(b.a) = a.(a+b) = (a+b).a = a+b.a = a

Montrons par exemple : a+(b.a) = a

a+(b.a)= a+a.b = a.1+a.b = a.(1+b) = a.1 = a

Le reste se montrant de la même façon.

Cette algèbre est utile pour décrire et étudier les schémas électroniques, mais elle sert aussi dans d'autres domaines que l'électricité. Elle est étudiée ici parce que les ordinateurs actuels sont basés sur des composants électroniques. Nous allons descendre un peu plus bas dans la réalisation interne du cœur d'un ordinateur, afin d'aboutir à la construction d'un additionneur en binaire dans l'UAL.

Tables de vérité des trois opérateurs :

x	y	$\overline{x}$	$x \,.\, y$	$x + y$
1	**1**	**0**	**1**	**1**
1	**0**	**0**	**0**	**1**
0	**1**	**1**	**0**	**1**
0	**0**	**1**	**0**	**0**

2. Circuits booléens ou logiques

Nous représentons par une variable booléenne x ∈ {**0**,**1**} le passage d'un courant électrique.

Lorsque x = **0**, nous dirons que x est à l'état 0 (**le courant ne passe pas**)

Lorsque x = **1**, nous dirons que x est à l'état 1 (**le courant passe**)

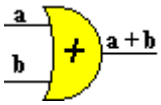
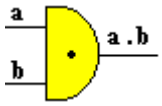
Une telle variable booléenne permet ainsi de visualiser, sous forme d'un bit d'information (0,1) le comportement d'un composant physique laissant ou ne laissant pas passer le courant.

Nous ne nous préoccuperons pas du type de circuits électriques permettant de construire un circuit logique (les composants électriques sont basés sur les circuits intégrés). Nous ne nous intéresserons qu'à la fonction logique (booléenne) associée à un tel circuit.

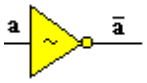
En fait un circuit logique est un opérateur d'une algèbre de Boole c'est-à-dire une combinaison de symboles de l'algèbre {**0,1**}, **.** ,+, $\overline{x}$).

3.1 Principaux circuits

Nous proposons donc 3 circuits logiques de base correspondant aux deux lois internes et à l'opérateur de complémentation involutif.

Le circuit OU associé à la loi " + " :  La table de vérité de ce circuit est celle de l'opérateur +	*Le circuit ET associé à la loi "•" :*  La table de vérité de ce circuit est celle de l'opérateur •

Le circuit NON associé à la loi " $\overline{\overline{x}}$ " :



la table de vérité est celle de l'opérateur involutif $\overline{\overline{x}}$

On construit deux circuits classiques à l'aide des circuits précédents :

L'opérateur XOR = " ou exclusif " :

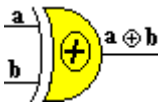
$a \oplus b = \overline{a}.b + a.\overline{b}$	dont voici le schéma : 

Table de vérité du ou exclusif :

a	b	$a \oplus b$
1	**1**	**0**
1	**0**	**1**
0	**1**	**1**
0	**0**	**0**

L'opérateur NAND (le NON-ET):

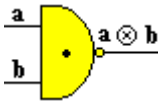
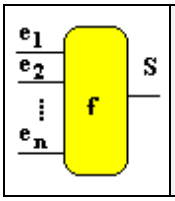
$a \otimes b = \overline{a.b} = \overline{a} + \overline{b}$	dont voici le schéma : 

Table de vérité du Nand :

a	b	$a \otimes b$
1	**1**	**0**
1	**0**	**1**
0	**1**	**1**
0	**0**	**1**

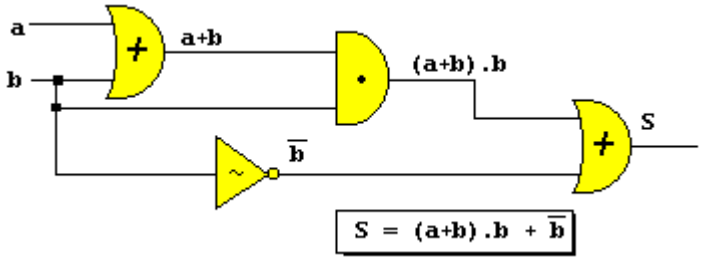
3.2 Fonction logique associée à un circuit



Un circuit logique est un système de logique séquentielle où la valeur de sortie S (état de la variable booléenne **S** de sortie) dépend des valeurs des entrées $\mathbf{e_1,e_2,...,e_n}$ (états des variables booléennes d'entrées e_i). Sa valeur de sortie est donc une fonction $\mathbf{S = f(e_1,e_2,...,e_n)}$.

Pour calculer la fonction **f** à partir d'un schéma de circuits logiques, il suffit d'indiquer à la sortie de chaque opérateur (circuit de base) la valeur de l'expression booléenne en cours. Puis, à la fin, nous obtenons une expression booléenne que l'on simplifie à l'aide des axiomes ou des théorèmes de l'algèbre de Boole.

Exemple :



En simplifiant S : $(a+b).b+\overline{b} = b + \overline{b}$ (formule d'absorbtion)
$b + \overline{b} = 1$.

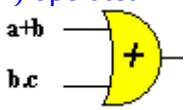
3.3 Circuit logique associé à une fonction

A l'inverse, la création de circuits logiques à partir d'une fonction booléenne **f** à n entrées est aussi simple. Il suffit par exemple, dans la fonction, d'exprimer graphiquement chaque opérateur par un circuit, les entrées étant les opérandes de l'opérateur. En répétant l'action sur tous les opérateurs, on construit un graphique de circuit logique associé à la fonction **f**.

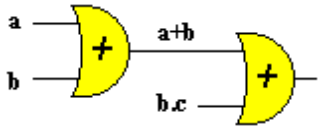
Exemple : Soit la fonction f de 3 variables booléennes, $f(a,b,c) = (a+b)+(b \bullet c)$

Construction progressive du circuit associé.

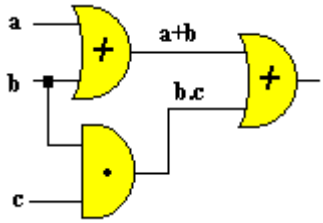
1°) opérateur " + " :



2°) branche supérieure de l'opérateur " + " :



3°) branche inférieure de l'opérateur " + " :



3.4 Additionneur dans l'UAL

a) Demi-additionneur

Reprenons les tables de vérités du " ⊕ " (*Xor*), du " + " et du " • " et adjoignons la table de calcul de l'addition en numération binaire.

Tout d'abord les tables comparées des opérateurs booléens :

a	b	$a \oplus b$	$a+b$	$a.b$
1	1	0	1	1
1	0	1	1	0
0	1	1	1	0
0	0	0	0	0

Rappelons ensuite la table d'addition en numération binaire :

+	**0**	**1**
0	0	1
1	1	0(1)

*0(1) représente la retenue **1** à reporter.*

En considérant une addition binaire comme la somme à effectuer sur deux mémoires à un bit, nous observons dans l'addition binaire les différentes configurations des bits concernés (notés ***a*** et ***b***).

Nous aurons comme résultat un bit de ***somme*** et un bit de ***retenue*** :

bit ***a***	bit ***b***	bit ***somme***	bit de ***retenue***
1 +	1 =	0	1
1 +	0 =	1	0
0 +	1 =	1	0
0 +	0 =	0	0

Si l'on compare avec les tables d'opérateurs booléens, on s'aperçoit que l'opérateur "⊕" (*Xor*) fournit en sortie les mêmes configurations que le bit de somme, et que l'opérateur "•" (*Et*) délivre en sortie les mêmes configurations que le bit de retenue.

Il est donc possible de simuler une addition binaire (arithmétique binaire) avec les deux opérateurs "⊕" et "•". Nous venons de construire un demi-additionneur ou additionneur sur un bit. Nous pouvons donc réaliser le circuit logique simulant la fonction complète d'addition binaire, nous l'appellerons " additionneur binaire "(somme arithmétique en binaire de deux entiers en binaire).

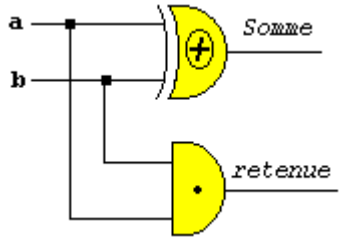


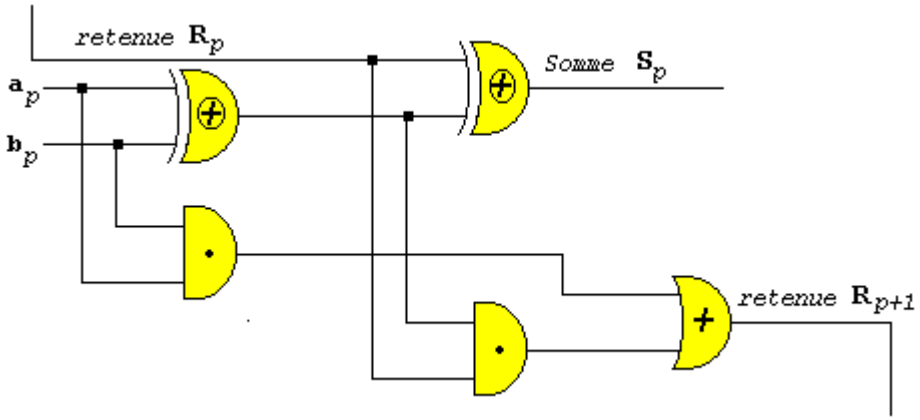
schéma logique d'un demi-additionneur

b)Additionneur complet

Une des constructions les plus simples et la plus pédagogique d'un additionneur complet est de connecter entre eux et en série des demi-additionneurs (additionneurs en cascade). Il existe une autre méthode dénommée " diviser pour régner " pour construire des additionneurs complets plus rapides à l'exécution que les additionneurs en cascade. Toutefois un additionneur en cascade pour UAL à 32 bits, utilise 2 fois moins de composants électroniques qu'un additionneur diviser pour régner.

Nous concluons donc qu'une UAL n'effectue en fait que des opérations logiques (algèbre de Boole) et simule les calculs binaires par des combinaisons d'opérateurs logiques

Soient a et b deux nombres binaires à additionner dans l'UAL. Nous supposons qu'ils sont stockés chacun dans une mémoire à n bits. Nous notons $\mathbf{a_p}$et $\mathbf{b_p}$ leur bit respectif de rang p. Lors de l'addition il faut non seulement additionner les bits $\mathbf{a_p}$ et $\mathbf{b_p}$ à l'aide d'un demi-aditionneur, mais aussi l'éventuelle retenue notée **Rp** provenant du calcul du rang précédent.

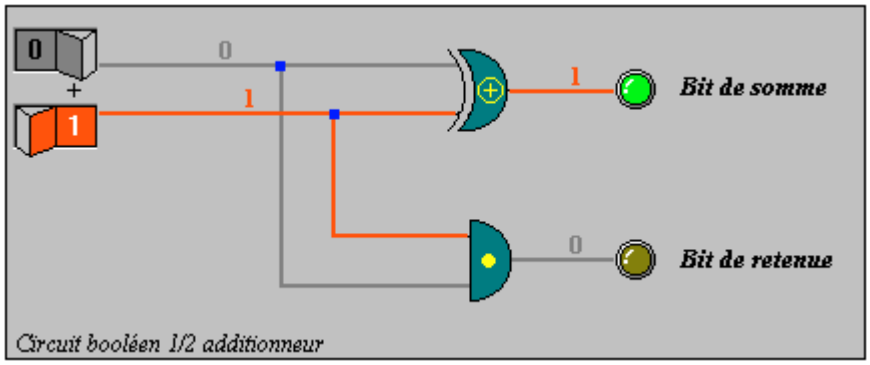


additionneur en cascade (addition sur le bit de rang p)

On réadditionne **Rp** à l'aide d'un demi-additionneur à la somme de $\mathbf{a_p}$et $\mathbf{b_p}$et l'on obtient le bit de somme du rang p noté **Sp**. La propagation de la retenue $\mathbf{Rp_{+1}}$ est faite par un " ou " sur les deux retenues de chacun des demi-additionneurs et passe au rang p+1. Le processus est itératif sur tous les n bits des mémoires contenant les nombres a et b.

Soit un exemple fictif de réalisation d'un demi-additionneur simulant l'addition binaire suivante :

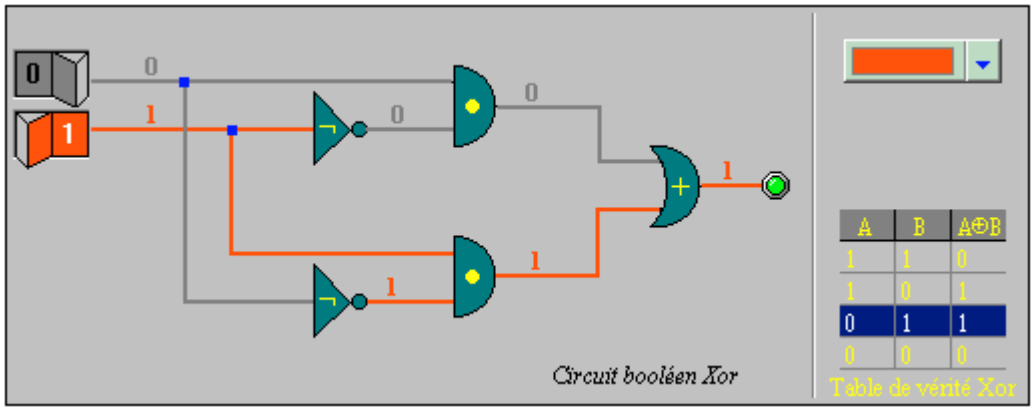
0 + 1 = 1. Nous avons figuré le passage ou non du courant à l'aide de deux interrupteurs (valeur = 1 indique que l'interrupteur est fermé et que le courant passe, valeur = 0 indique que l'interrupteur est ouvert et que le courant ne passe pas)



Le circuit « **et** » fournit le bit de retenue soit : 0 •1 = 0

Le circuit « **Xor** » fournit le bit de somme soit : 0 ⊕ 1 = 1

Nous figurons le détail du circuit Xor du schéma précédent lorsqu'il reçoit le courant des deux interrupteurs précédents dans la même position (l'état électrique correspond à l'opération 0 ⊕ 1 = 1)

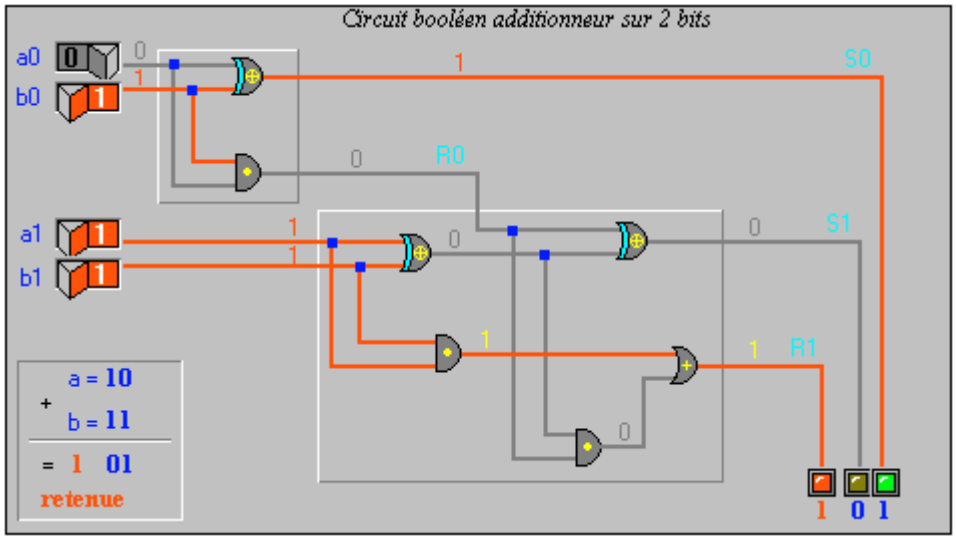


Si l'UAL effectue des additions sur 32 bits, il y aura 32 circuits comme le précédent, tous reliés en série pour la propagation de la retenue.

Un exemple d'additionneur sur deux mémoires **a** et **b** à 2 bits contenant respectivement les nombres 2 et 3 :

a = | **1** | **0** |

b = | **1** | **1** |



Les 4 interrupteurs figurent le passage du courant sur les bits de même rang des mémoires **a=2** et **b=3**, le résultat obtenu est la valeur attendue soit 2+3 = 5.

1.3 Codage numération

Plan du chapitre:

1. Codage de l'information

1.1 Codage en général : le pourquoi
1.2 Codage des caractères : code ASCII
1.3 Codage des nombres entiers : numération
1.4 Les entiers dans une mémoire à n+1 bits
1.5 Codage des nombres entiers
1.6 Un autre codage des nombres entiers

2. Numération

2.1 Opérations en binaire
2.2 Conversions base quelconque ⇔ décimal
2.3 Exemple de conversion décimal → binaire
2.4 Exemple de conversion binaire → décimal
2.5 Conversion binaire → hexadécimal
2.6 Conversion hexadécimal → binaire

1. Codage de l'information

1.1 Codage en général : le pourquoi

- Dans une machine, toutes les informations sont codées sous forme d'une suite de "0" et de "1" (langage binaire). Mais l'être humain ne parle généralement pas couramment le langage binaire.
- Il doit donc tout "traduire" pour que la machine puisse exécuter les instructions relatives aux informations qu'on peut lui donner.
- Le codage étant une opération purement humaine, il faut produire des algorithmes qui permettront à la machine de traduire les informations que nous voulons lui voir traiter.

Le **codage** est une opération établissant une bijection entre une **information** et une **suite de " 0 " et de " 1 "** qui sont représentables en machine.

1.2 Codage des caractères : code ASCII

Parmi les codages les plus connus et utilisés, le codage **ASCII** (American Standard Code for Information Interchange)étendu est le plus courant (version **ANSI** sur Windows).

Voyons quelles sont les nécessités minimales pour l'écriture de documents alphanumériques simples dans la civilisation occidentale. Nous avons besoin de :

Un alphabet de lettres minuscules ={a, b, c,...,z}
soient 26 caractères

Un alphabet de lettres majuscules ={A,B,C,...,Z}
soient 26 caractères

Des chiffres {0,1,...,9}
soient 10 caractères

Des symboles syntaxiques {? , ; (" ...
au minimum *10 caractères*

Soit un total minimal de *72 caractères*

Si l'on avait choisi un code à 6 bits le nombre de caractères codables aurait été de $\mathbf{2^6 = 64}$ (tous les nombres binaires compris entre **000000** et **111111**), nombre donc insuffisant pour nos besoins.

Il faut au minimum 1 bit de plus, ce qui permet de définir ainsi $\mathbf{2^7 = 128}$ nombres binaires différents, autorisant alors le codage de 128 caractères.

Initialement le code **ASCII** est un code à 7 bits (**2^7 = 128 caractères**). Il a été étendu à un code sur 8 bits (**2^8 = 256 caractères**) permettant le codage des caractères nationaux (en France les caractères accentués comme : ù,à,è,é,â,...etc) et les caractères semi-graphiques.

Les pages HTML qui sont diffusées sur le réseau Internet sont en code ASCII 8 bits.

Un codage récent dit " universel " est en cours de diffusion : il s'agit du codage **Unicode** sur 16 bits (**2^{16} = 65536 caractères**).

De nombreux autres codages existent adaptés à diverses solution de stockage de l'information (DCB, EBCDIC,...).

1.3 Codage des nombres entiers : numération

Les nombres entiers peuvent être codés comme des caractères ordinaires. Toutefois les codages adoptés pour les données autres que numériques sont trop lourds à manipuler dans un ordinateur. Du fait de sa constitution, un ordinateur est plus " habile " à manipuler des nombres écrits en numération binaire (qui est un codage particulier).

Nous allons décrire trois modes de codage des entiers les plus connus.

Nous avons l'habitude d'écrire nos nombres et de calculer dans le système décimal. Il s'agit en fait d'un cas particulier de numération en base 10.

Il est possible de représenter tous les nombres dans un système à base b (b entier, b1). Nous ne présenterons pas ici un cours d'arithmétique, mais seulement les éléments nécessaires à l'écriture dans les deux systèmes les plus utilisés en informatique : le binaire (b=2) et l'hexadécimal (b=16).

Lorsque nous écrivons **5876** en base 10, la position des chiffres 5,8,7,6 indique la puissance de 10 à laquelle ils sont associés :

5 est associé à 10^3
8 est associé à 10^2
7 est associé à 10^1
6 est associé à 10^0

Il en est de même dans une base b quelconque (b=2, ou b=16). Nous conviendrons de mentionner la valeur de la base au dessus du nombre afin de savoir quel est son type de représentation.

Soit $\overline{x_n x_{n-1} \cdots x_0}^{\,b}$ un nombre **x** écrit en base b avec n+1 symboles.

- " $\mathbf{x_k}$ " est le symbole associé à la puissance " $\mathbf{b^k}$ "
- " $\mathbf{x_k}$ " $\in$ {0,1, ... , b-1}

Lorsque b=2 (numération binaire)

Chaque symbole du nombre x, " $\mathbf{x_k}$ " $\in$ {0,1}; autrement dit les nombres binaires sont donc écrits avec des 0 et des 1, qui sont représentés physiquement par des bits dans la machine.

Voici le schéma d'une mémoire à n+1 bits (au minimum 8 bits dans un micro-ordinateur) :

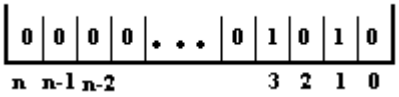
Les cases du schéma représentent les bits, le chiffre marqué en dessous d'une case, indique la puissance de 2 à laquelle est associé ce bit (on dit aussi le ***rang*** du bit).

Le bit de rang 0 est appelé le bit de **poids faible**.
Le bit de rang n est appelé le bit de **poids fort**.

1.4 Les entiers dans une mémoire à n+1 bits : binaire pur

Ce codage est celui dans lequel les nombres entiers **naturels** sont écrits en numération binaire (en base b=2).

Le nombre " dix " s'écrit **10** en base b=10, il s'écrit **1010** en base b=2. Dans la mémoire ce nombre dix est codé en binaire ainsi:



Une mémoire à n+1 bits (n>0), permet de représenter sous forme binaire (en binaire pur) tous les entiers naturels de l'intervalle **[0 , 2^{n+1} -1]**.

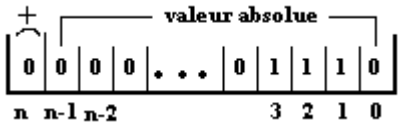
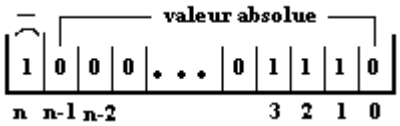
- soit pour n+1=8 bits, tous les entiers de l'intervalle **[0 , 255]**
- soit pour n+1=16 bits, tous les entiers de l'intervalle **[0 , 65535]**

1.5 Codage des nombres entiers : binaire signé

Ce codage permet la représentation des nombres entiers relatifs.

Dans la représentation en binaire signé, le bit de poids fort *(bit de rang n associé à 2^n)* sert à représenter le signe (0 pour un entier positif et 1 pour un entier négatif), les **n** autres bits représentent la valeur absolue du nombre en binaire pur.

Exemple du codage en binaire signé des nombres **+14** et **-14** :

 *+14 est représenté par 0000...01110*	 *-14 est représenté par 1000...01110*

Une mémoire à n+1 bits (n>0), permet de représenter sous forme binaire (en binaire signé) tous les entiers naturels de l'intervalle **[- (2^n - 1) , (2^n -1)]**

- soit pour n+1=8 bits, tous les entiers de l'intervalle **[-127 , 127]**
- soit pour n+1=16 bits, tous les entiers de l'intervalle **[-32767 , 32767]**

Le nombre zéro est représenté dans cette convention (dites du zéro positif) par : **0**000...00000

Remarque : Il reste malgré tout une configuration mémoire inutilisée : **1**000...00000. Cet état binaire ne représente à priori aucun nombre entier ni positif ni négatif de l'intervalle [- (2^n - 1) , (2^n -1)]. Afin de ne pas perdre inutilement la configuration " **1**000...00000 ", les informaticiens ont décidé que cette configuration représente malgré tout un nombre négatif parce que le bit de signe est 1, et en même temps la puissance du bit contenant le "1", donc par convention -2^n.

L'intervalle de représentation se trouve alors augmenté d'un nombre :
il devient :[-2^n ,2^n -1]

- soit pour n+1=8 bits, tous les entiers de l'intervalle **[-128 , 127]**
- soit pour n+1=16 bits, tous les entiers de l'intervalle **[-32768 , 32767]**

Ce codage n'est pas utilisé tel quel, il est donné ici à titre pédagogique. En effet le traitement spécifique du signe coûte cher en circuits électroniques et en temps de calcul. C'est une version améliorée qui est utilisée dans la plupart des calculateurs : elle se nomme le complément à deux.

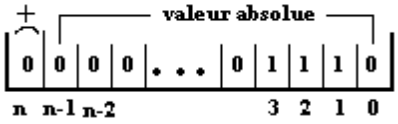
1.6 Un autre codage des nombres entiers : complément à deux

Ce codage, purement conventionnel et très utilisé de nos jours, est dérivé du binaire signé ; il sert à représenter en mémoire les entiers relatifs.

Comme dans le binaire signé, la mémoire est divisée en deux parties inégales; le bit de poids fort représentant le signe, le reste représente la valeur absolue avec le codage suivant :

Supposons que la mémoire soit à n+1 bits, soit **x** un entier relatif à représenter :

si **x > 0,** alors c'est la convention en *binaire signé* qui s'applique (le bit de signe vaut 0, les n bits restants codent le nombre), soit pour le nombre +14 :

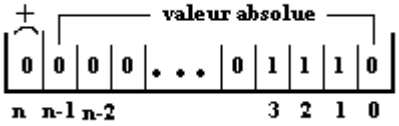


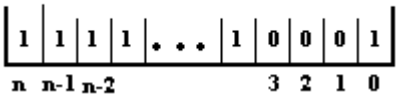
+14 est représenté par 0000...01110

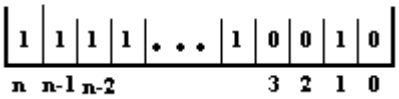
si **x < 0**, alors (3 étapes à suivre)

- On code la valeur absolue du nombre x, **|x|** en binaire signé.
- Puis l'on complémente tous les bits de la mémoire (complément à 1 ou complément restreint). Cette opération est un **non** logique effectué sur chaque bit de la mémoire.
- Enfin l'on additionne +**1** au nombre binaire de la mémoire (addition binaire).

Exemple, soit à représenter le nombre -14 en suivant les 3 étapes :

- 

codage de |-14|= 14
- 

complément à 1
- 

addition de 1

Le nombre -14 s'écrit donc en complément à 2 : 1111..10010.

Un des intérêts majeurs de ce codage est d'intégrer la soustraction dans l'opération de codage et de ne faire effectuer que des opérations simples et rapides (non logique, addition de 1).

Nous venons de voir que le codage utilisait essentiellement la représentation d'un nombre en binaire (la numération binaire) et qu'il fallait connaître les rudiments de l'arithmétique binaire. Le paragraphe ci-après traite de ce sujet.

2. Numération

Ce paragraphe peut être ignoré par ceux qui connaissent déjà les éléments de base des calculs en binaire et des conversions binaire-décimal-hexadécimal, dans le cas contraire, il est conseillé de le lire.

Pour des commodités d'écriture, nous utilisons la notation indicée pour représenter la base d'un nombre en parallèle de la représentation avec la barre au dessus. Ainsi 145_{10} signifie le nombre 145 en base dix; 1101011_2 signifie 1101011 en binaire.

2.1 Opérations en binaire

Nous avons parlé d'addition en binaire ; comme dans le système décimal, il nous faut connaître les tables d'addition, de multiplication, etc... afin d'effectuer des calculs dans cette base. Heureusement en binaire, elles sont très simples :

Addition

+	0	1
0	0	1
1	1	0(1)

Multiplication

*	0	1
0	0	0
1	0	1

0(1) représente la retenue 1 à reporter.

*Exemples de calculs (109+19=**128_{10}** =10000000_2) et (22x5=**110**) :*

addition ***multiplication***

$$\begin{array}{r} 1101101 \\ +\quad 10011 \\ \hline 10000000_2 = 128_{10} \end{array}$$

$$\begin{array}{r} 10110 \\ \mathbf{x}\quad 101 \\ \hline 10110 \\ 10110.. \\ \hline 1101110_2 = 110_{10} \end{array}$$

Vous noterez que le procédé est identique à celui que vous connaissez en décimal. En hexadécimal (b=16) il en est de même. Dans ce cas les tables d'opérateurs sont très longues à apprendre.

Etant donné que le système classique utilisé par chacun de nous est le système décimal, nous nous proposons de fournir d'une manière pratique les conversions usuelles permettant d'écrire les diverses représentations d'un nombre entre les systèmes décimal, binaire et hexadécimal.

2.2 Conversions base quelconque ⇔ décimal

Voici ci-dessous un rappel des méthodes générales permettant de convertir un nombre en base b (b>1)en sa représentation décimale et réciproquement.

A) Soit $\overline{x_n x_{n-1} \cdots x_0}^{\,b}$ un nombre écrit en base b.

Pour le convertir en décimal (base 10), il faut :

- **convertir chaque symbole x_k en son équivalent** a_k en base 10, nous obtenons ainsi la suite de chiffres : $a_n,\ldots,a_0$

Exemple, soit b=13, ***les symboles de la base 13 sont :* { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C }**

Si le chiffre $\mathbf{x_k}$ de rang k du nombre s'écrit **C**, son équivalent en base 10 est a_k=12

- réécrire le nombre comme une somme :

$$\overline{x_n x_{n-1} \cdots x_0}^{\,b} \equiv \sum_{k=0}^{n} x_k\, b^k \underset{\text{(en base 10)}}{\rightarrow} \sum_{k=0}^{n} \alpha_k\, b^k$$

- effectuer tous les calculs en base 10 (somme, produit, puissance).

Exemple : $\overline{2AB8}^{\,13}$ **à convertir en base10 (b=13)**

Le nombre $\overline{\underset{3}{2}\ \underset{2}{A}\ \underset{1}{B}\ \underset{0}{8}}^{\,13} \equiv 2.13^3 + 10.13^2 + 11.13^1 + 8.13^0$

$= 4394 + 1690 + 143 + 8 = \overline{6235}^{\,10}$

B) Soit " **a** " un nombre écrit décimal et à représenter en base **b** :

La méthode utilisée est un algorithme fondé sur la division euclidienne.

- **Si a < b**, il n'a pas besoin d'être converti.
- **Si a = b**, on peut diviser a par b. Et l'on divise successivement les différents quotients $\mathbf{q_k}$ obtenus par la base b.

De manière générale on aura :

$\mathbf{a = b^k.r_k + b^{k-1}.r_{k-1} + \ldots + b.r_1 + r_0}$ (où $\mathbf{r_i}$ est le reste de la division de a par b).

En remplaçant chaque $\mathbf{r_i}$ par son symbole équivalent $\mathbf{p_i}$ en base **b**, nous obtenons :

$$a = \overline{p_k\ p_{k-1}\ \cdots\ p_1\ p_0}^{\,b}$$

Cet algorithme permet d'obtenir une représentation de a dans la base b.

Exemple :

$\overline{6235}^{10}$ **à convertir en base b=13**

les symboles de la base 13 sont: {0,1,2,3,4,5,6,7,8,9,A,B,C}

```
6235   |  13
       |------ q0
  8    |  479   |  13
r0             |------ q1
          11   |  36   |  13
        r1             |------ q2
                  10   |   2
                r2         r3 = q2
```

Les p_i *équivalents en base 13 sont:*

$r_0 = 8$	$\rightarrow p_0 = 8$	
$r_1 = 11$	$\rightarrow p_1 = B$	
$r_2 = 10$	$\rightarrow p_2 = A$	
$r_3 = 2$	$\rightarrow p_3 = 2$	Donc $\mathbf{6235}_{10} = \mathbf{2AB8}_{13}$

Dans les deux paragraphes suivants nous allons expliciter des exemples pratiques de ces méthodes dans le cas où la base est 2 (binaire).

2.3 Exemple de conversion décimal → binaire

Soit le nombre décimal $\mathbf{35}_{10}$, appliquons l'algorithme précédent afin de trouver les restes successifs :

```
  35    |  2
        |------
r0 = 1  |  17   |  2
                |------
      r1 = 1    |  8   |  2
                       |------
          r2 =  0      |  4   |  2
                             |------
               r3 =  0       |  2   |  2
                                   |------
                        r4 = 0     |  1 = r5
```

Donc : $\mathbf{35}_{10} = \mathbf{100011}_2$

2.4 Exemple de conversion binaire → décimal

Soit le nombre binaire : **1101101**$_2$
sa conversion en décimal est immédiate :

1101101$_2$ = $2^6 + 2^5 + 2^3 + 2^3 + 2^2 + 1 = 64+32+8+4+1 =$ **109**$_{10}$

Les informaticiens, pour des raisons de commodité (manipulations minimales de symboles), préfèrent utiliser l'hexadécimal plutôt que le binaire. L'humain, contrairement à la machine, a quelques difficultés à fonctionner sur des suites importantes de 1 et de 0. Ainsi l'hexadécimal (sa base b=2^4 étant une puissance de 2) permet de diviser, en moyenne, le nombre de symboles par un peu moins de 4 par rapport au même nombre écrit en binaire. C'est l'unique raison pratique qui justifie son utilisation ici.

2.5 Conversion binaire → hexadécimal

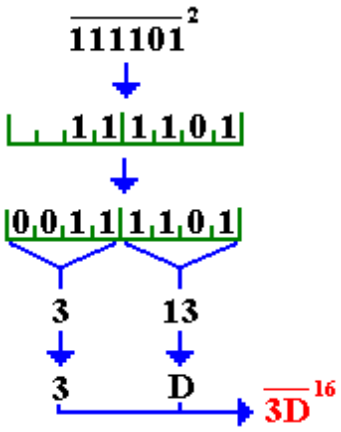
Nous allons détailler l'action de conversion en 6 étapes pratiques :

- Soit a un nombre écrit en **base 2** (***étape 1***).
- On décompose ce nombre par tranches de 4 bits à partir du bit de poids faible (***étape 2***).
- On complète la dernière tranche (celle des bits de poids forts)par des 0 s'il y a lieu (***étape 3***).
- On convertit chaque tranche en son symbole de la **base 16**(***étape 4***).
- On réécrit à sa place le nouveau symbole par changements successifs de chaque groupe de 4 bits,(***étape 5***).
- Ainsi, on obtient le nombre écrit en hexadécimal (***étape 6***).

Exemple :

Soit le nombre **111101**$_2$

à convertir en héxadécimal.



Résultat obtenu :

111101$_2$ = **3D**$_{16}$

2.6 Conversion hexadécimal → binaire

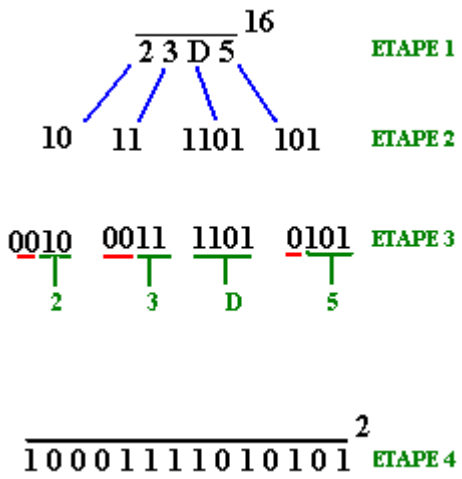
Cette conversion est l'opération inverse de la précédente. Nous allons la détailler en 4 étapes :

- Soit a un nombre écrit en **base 16** (***ETAPE 1***).
- On convertit chaque symbole hexadécimal du nombre en son écriture binaire (nécessitant au plus 4 bits) (***ETAPE 2***).
- Pour chaque tranche de 4 bits, on complète les bits de poids fort par des 0 s'il y a lieu (***ETAPE 3***).
- Le nombre " **a** " écrit en binaire est obtenu en regroupant toutes les tranches de 4 bits à partir du bit de poids faible, sous forme d'un seul nombre binaire(***ETAPE 4***).

Exemple :

Soit le nombre **23D5**$_{16}$

à convertir en binaire.



Résultat obtenu :

23D5$_{16}$ = **10001111010101**$_{2}$

1.4 Formalisation de la notion d'ordinateur

Plan du chapitre:

1. Machine de Turing théorique

1.1 Définition : machine de Turing
1.2 Définition : Etats de la machine
1.3 Définition : Les règles de la machine

2. La Machine de Turing physique

2.1 Constitution interne
2.2 Fonctionnement
2.3 Exemple : machine de Turing arithmétique
2.4 Machine de Turing informatique

1. La Machine de Turing théorique

Entre 1930 et 1936 le mathématicien anglais A.Turing invente sur le papier une machine fictive qui ne pouvait effectuer que 4 opérations élémentaires que nous allons décrire. Un des buts de Turing était de faire résoudre par cette " machine " des problèmes mathématiques, et d'étudier la classe des problèmes que cette machine pourrait résoudre.

Définitions et notations (modèle déterministe)
Soit A un ensemble fini appelé *alphabet* défini ainsi :

$A = \{ a_1 ,..., a_n \}$ $(A \neq \emptyset)$
Soit $\Omega = \{D,G\}$ une paire

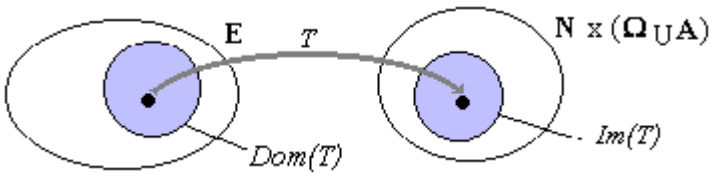
1.1 Définition : machine de Turing

Nous appellerons machine de Turing toute application T :

$T : E \rightarrow \mathbf{N} \times (\Omega \cup A)$
où E est un ensemble fini non vide : $E \subset \mathbf{N} \times A$

1.2 Définition : Etats de la machine

Nous appellerons E_t ensemble des états intérieurs de la machine T:
$E_t = \{ q_i \in \mathbf{N} , (\exists a_i \in A / (q_i , a_i) \in Dom(T)) \text{ où } (\exists x \in \Omega / (q_i , x) \in Im(T)) \}$



Dom(T) : domaine de définition de T.
Im(T) : image de T (les éléments T(a) de $\mathbf{N} \times (\Omega \cup A)$, pour $a \in E$)

Comme E est un ensemble fini, E_t est nécessairement un ensemble fini, donc il y a un nombre fini d'états intérieurs notés q_i.

1.3 Définition : Les règles de la machine

Nous appellerons " ensemble des règles " de la machine T, le graphe G de l'application T. Une règle de T est un élément du graphe G de T.

On rappelle que le graphe de T est défini comme suit :

$G = \{(a,b) \in E \times [E_t \times (\Omega \cup A)] \ / \ b = T(a) \}$

- **Notation** : afin d'éviter une certaine lourdeur dans l'écriture nous conviendrons d'écrire les règles en omettant les virgules et les parenthèses.
- **Exemple** : la règle $((q_i,a),(q_k,b))$ est notée : $q_i\, a\, q_k\, b$

2. La Machine de Turing physique

2.1 Constitution interne

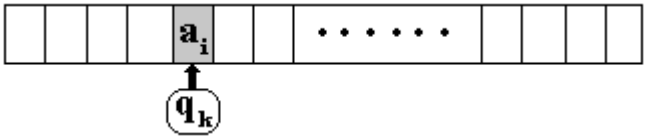
Nous construisons une machine de Turing physique constituée de :

- Une boîte notée UC munie d'une tête de lecture-écriture et d'un registre d'état.
- Un ruban de papier supposé sans limite vers la gauche et vers la droite.
- Sur le ruban se trouvent des cases contigües contenant chacune un seul élément de l'alphabet A.
- La tête de lecture-écriture travaille sur la case du ruban située devant elle ; elle peut lire le contenu de cette case ou effacer ce contenu et y écrire un autre élément de A.
- Il existe un dispositif d'entraînement permettant de déplacer la tête de lecture-écriture d'une case vers la **Droite** ou vers la **Gauche**.
- Dans la tête lecture-écriture il existe une case spéciale notée ***registre d'état***, qui sert à recevoir un élément q_i de E_t.

Cette machine physique est une représentation virtuelle d'une machine de Turing théorique T, d'alphabet A, dont l'ensemble des états est E_t, dont le graphe est donné ci-après :

$G = \{(a,b) \in E \times [E_t \times (\Omega \cup A)] \ / \ b = T(a) \}$

Donnons une visualisation schématique d'une telle machine en cours de fonctionnement. La tête de lecture/écriture pointe vers une case contenant l'élément $\mathbf{a_i}$ de A, le registre d'état ayant la valeur $\mathbf{q_k}$:



2.2 Fonctionnement

Départ :

On remplit les cases du ruban d'éléments a_i de A.

On met la valeur " q_k " dans le registre d'état.

On positionne la tête sur une case contenant " a_i ".

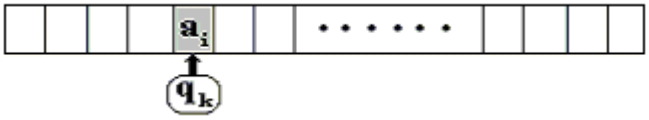
Actions : (la machine se met en marche)

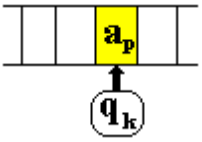
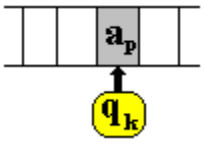
La tête lit le " a_i ". L'UC dont le registre d'état vaut " q_k ", cherche dans la liste des règles si le couple $(q_k , a_i) \in Dom(T)$.

Si la réponse est *négative* on dit que la machine " bloque " (elle s'arrête par blocage).

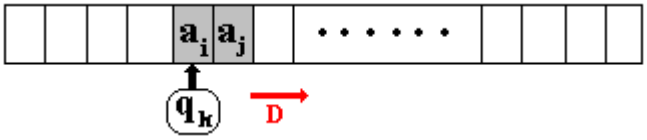
Si la réponse est *positive* alors le couple (q_k , a_i) a une image unique (machine déterministe)que nous notons $(q_n , \mathbf{y})$. Dans ce couple, **y** ne peut prendre que l'un des 3 types de valeurs possibles $\mathbf{a_p}$, **D** , **G** :

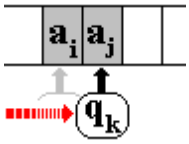
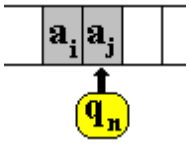
- ***a)*** **soit y=a_p ,dans ce cas la règle est donc de la forme $q_k\, a_i\, q_n\, a_p$**



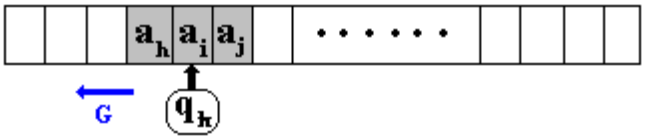
a.1) L'UC fait effacer le $\mathbf{a_i}$ dans la case et le remplace par l'élément $\mathbf{a_p}$.	
a.2) L'UC écrit $\mathbf{q_n}$ dans le registre d'état en remplacement de la valeur $\mathbf{q_k}$.	

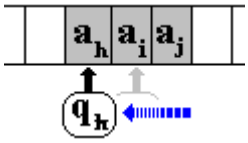
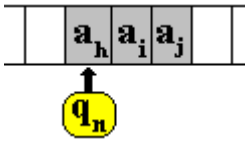
- ***b)*** **soit y = D , ici la règle est donc de la forme $q_k\, a_i\, q_n\, D$**



b.1) L'UC fait déplacer la tête vers la droite d'une case.	
b.2) L'UC écrit $\mathbf{q_n}$ dans le registre d'état en remplacement de la valeur $\mathbf{q_k}$.	

- ***c)*** **soit y = G , dans ce cas la règle est donc de la forme** $q_k\, a_i\, q_n\, G$



c.1) L'UC fait déplacer la tête vers la gauche d'une case.	
c.2) L'UC écrit q_n dans le registre d'état en remplacement de la valeur $\mathbf{q_k}$.	

Puis la machine continue à fonctionner en recommençant le cycle des actions depuis le début : lecture du nouvel élément a_k etc...

2.3 Exemple : machine de Turing arithmétique

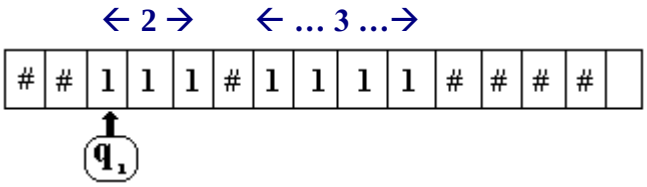
Nous donnons ci-dessous une machine T_1 additionneuse en arithmétique unaire.

A = {#,1}

Ω ={D,G}

un entier n est représenté par n+1 symboles " **1** " consécutifs (de façon à pouvoir représenter " 0 " par un unique " 1 ").

Etat initial du ruban avant actions :



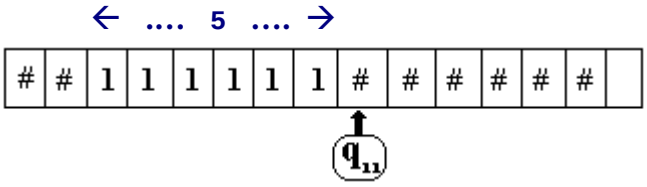
2 est représenté par ***111***
3 est représenté par ***1111***

Règles T_1: (application des règles suivantes pour simulation de 2+3)

q_1 **1** q_2**D**	q_6 **1** q_7**D**	q_{11} **1** q_{12}**#**
q_2 **1** q_3**D**	q_7 **1** q_8**D**	q_{12} **#** q_{13}**G**
q_3 **1** q_4**D**	q_8 **1** q_9**D**	q_{13} **1** q_{14}**#**
q_4 **#** q_5**1**	q_9 **1** q_{10}**D**	
q_5 **1** q_6**D**	q_{10} **#** q_{11}**G**	

En démarrant la machine sur la configuration précédente on obtient :

Etat final du ruban après actions : (Cette machine ne fonctionne que pour additionner 2 et 3)



.

Généralisation de la machine additionneuse

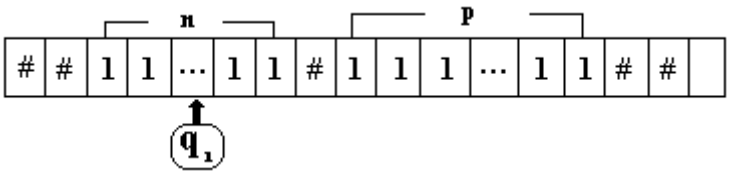
Il est facile de fournir une autre version plus générale **T_2** fondée sur la même stratégie et le même état initial permettant d'additionner non plus seulement les nombres 2 et 3, mais des nombres entiers quelconques **n** et **p**. Il suffit de construire des nouvelles règles.

Règles de T_2:

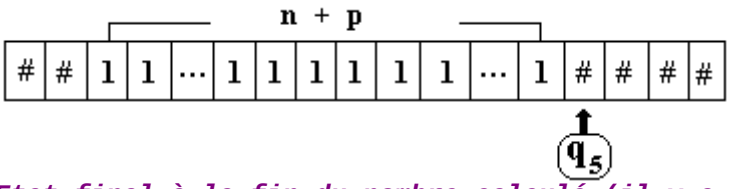
q_1 **1** q_1**D**	q_3 **1** q_3**#**
q_1 **#** q_2**1**	q_3 **#** q_4**G**
q_2 **1** q_2**D**	q_4 **1** q_5**#**
q_2 **#** q_3**G**	

Cette machine de Turing $\mathbf{T_2}$ appliquée à l'exemple précédent (addition de 2 et 3) laisse le ruban dans le même état final, mais elle est construite avec moins d'états intérieurs que la précédente.

En fait elle fonctionne aussi si la tête de lecture-écriture est positionnée sur n'importe lequel des éléments du premier nombre **n**. et les nombres **n** et **p** sont quelconques :



Etat initial sur le nombre de gauche



Etat final à la fin du nombre calculé (il y a n+p+1 symboles " 1 ")

Nous dirons que T_2 est plus " *puissante* " que T_1 au sens suivant :

- T_2 a moins d'états intérieurs que T_1 .
- T_2 permet d'additionner des entiers quelconques.
- Il est possible de démarrer l'état initial sur n'importe lequel des " 1 " du nombre de gauche.

On pourrait toujours en ce sens chercher une machine T_3 qui posséderait les qualités de T_2 , mais qui pourrait démarrer sur n'importe lequel des " 1 " de l'un ou l'autre des deux nombres **n** ou **p**, le lecteur est encouragé à chercher à écrire les règles d'une telle machine.

Nous voyons que ces machines sont capables d'effectuer des opérations, elles permettent de définir la classe des **fonctions calculables** (par machines de Turing).

Un ordinateur est fondé sur les principes de calcul d'une machine de Turing. J. Von Neumann a défini la structure générale d'un ordinateur à partir des travaux de A.Turing. Les éléments physiques supplémentaires que possède un ordinateur moderne n'augmentent pas sa puissance théorique. Les fonctions calculables sont les seules que l'on puisse implanter sur un ordinateur. Les périphériques et autres dispositifs auxiliaires extérieurs ou intérieurs n'ont pour effet que d'améliorer la " *puissance* " en terme de vitesse et de capacité. Comme une petite voiture de série et un bolide de formule 1 partagent les mêmes concepts de motorisation, de la même manière les différents ordinateurs du marché partagent les mêmes fondements mathématiques.

2.4 Machine de Turing informatique

Nous faisons évoluer la représentation que nous avons de la machine de Turing afin de pouvoir mieux la programmer, c'est à dire pouvoir écrire plus facilement les règles de fonctionnement d'une telle machine.

Nous définissons ce qu'est un algorithme pour une machine de Turing et nous proposons une description graphique de cet algorithme à l'aide de schémas graphiques symboliques décrivant des actions de base d'une machine de Turing.

A) Une machine de Turing informatique est dite normalisée au sens suivant :

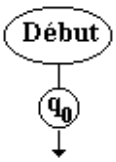
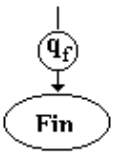
- L'alphabet A contient toujours le symbole " # "
- L'ensemble des états E contient toujours deux états distingués $\mathbf{q_0}$ (état initial) et $\mathbf{q_f}$ (état final).
- La machine démarre toujours à l'état initial $\mathbf{q_0}$.
- Elle termine sans blocage toujours à l'état $\mathbf{q_f}$.
- Dans les autres cas on dit que la machine " bloque ".

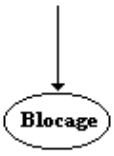
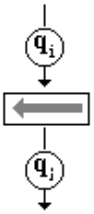
B)Algorithme d'une machine de Turing informatique

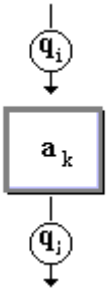
C'est l'ensemble des règles précises qui définissent un procédé de calcul destiné à obtenir en sortie un " **résultat** " déterminé à partir de certaines " **données** " initiales.

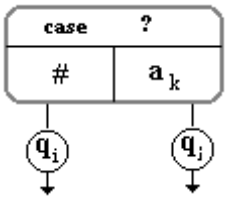
C) Algorithme graphique d'une machine de Turing

Nous utilisons cinq classes de symboles graphiques

	Positionne la tête de lecture sur le symbole voulu, met la machine à l'état initial q_0 et fait démarrer la machine.
	Signifie que la machine termine correctement son calcul en s'arrêtant à l'état final q_f .

	Aucune règle de la machine ne permettant la poursuite du fonctionnement, arrêt de la machine sur un blocage.
	Déplacer la tête d'une case vers la gauche (la tête de lecture se positionne sur la case d'avant la case actuelle contenant le symbole a_p). Correspond à la règle : $\mathbf{q_i\ a_p\ q_j\ G}$

	Correspond à l'action à exécuter dans la deuxième partie de la règle : $\mathbf{q_i\ a_p\ q_j\ a_k}$ (la machine écrit $\mathbf{a_k}$ dans la case actuelle et passe à l'état $\mathbf{q_j}$).

	Correspond à l'action à exécuter dans la première partie de la règle : $\mathbf{q_j\ a_k\ \ldots}$ ou $\mathbf{q_i\ \#\ \ldots}$ (la machine teste le contenu de la case actuelle et passe à l'état $\mathbf{q_j}$ ou à l'état $\mathbf{q_i}$ selon la valeur du contenu).

D) Organigramme d'une machine de Turing

On appelle organigramme d'une machine de Turing T, toute représentation graphique constituée de combinaisons des symboles des **cinq classes précédentes**.

Les règles de la forme $q_n\ \mathbf{a}_k\ q_n\ \mathbf{G}$ ou $q_n\mathbf{a}_k q_n\mathbf{D}$ se traduisent par des schémas " bouclés " ce qui donne des organigrammes non linéaires.

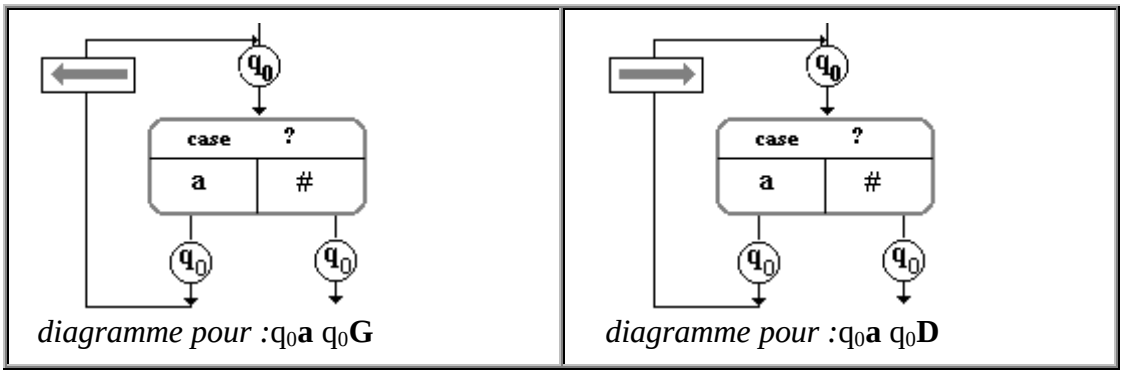
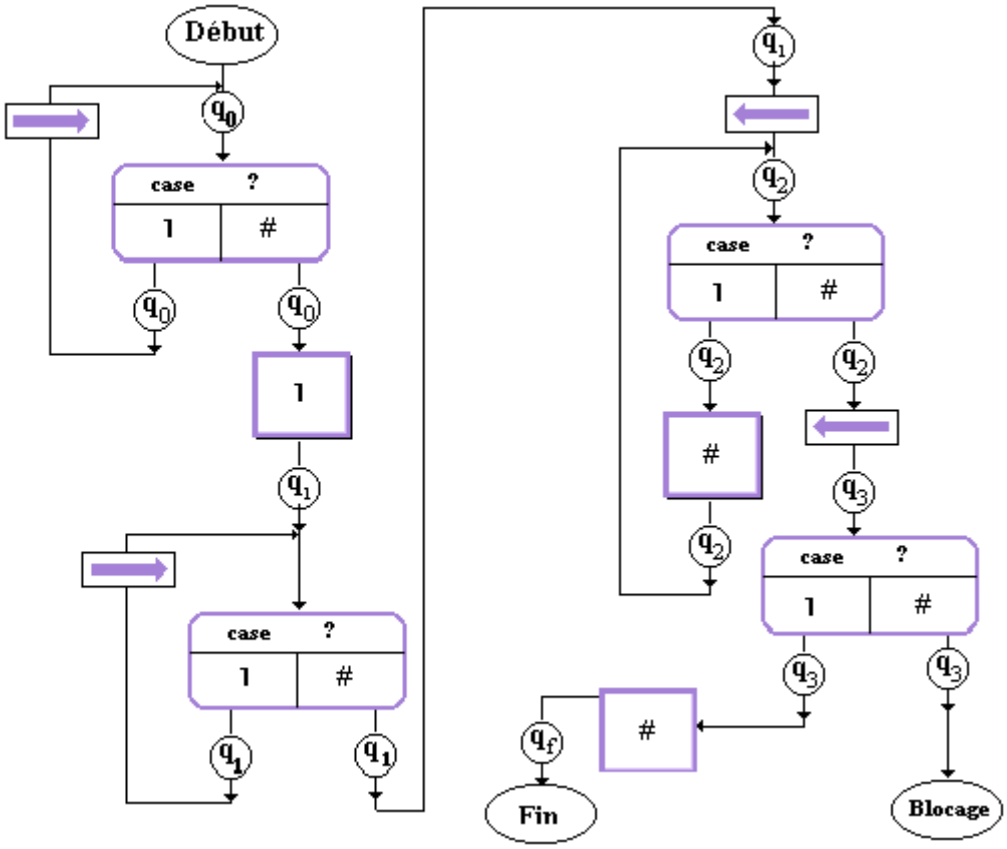


diagramme pour :q_0**a** q_0**G** | *diagramme pour :*q_0**a** q_0**D**

Exemple : organigramme de la machine T_2 normalisée(additionneuse n+p)



Règles de T_2 normalisée :

q_0 **1** q_0**D**	q_2 **1** q_2#
q_0 # q_1**1**	q_2 # q_3**G**
q_1 **1** q_1**D**	q_3 **1** q_f#
q_1 # q_2**G**	

Nous voyons que ce symbolisme graphique est un outil de description du mode de traitement de l'information au niveau machine. C'est d'ailleurs historiquement d'une façon semblable que les premiers programmeurs décrivaient leur programmes.

1.5 Architecture de l'ordinateur

Plan du chapitre:

Les principaux constituants

1.1 Dans l'Unité Centrale : l'unité de traitement
1.2 Dans l'Unité Centrale : l'unité de commande
1.3 Dans l'Unité Centrale : les Unités d'échange
1.4 Exemples de machine à une adresse : un micro-processeur
1.5 Les Bus
1.6 Schéma général d'une micro-machine fictive
1.7 Notion de jeu d'instructions-machine

2. Mémoires - Mémoire centrale

2.1 Mémoire
2.2 Les différents types de mémoires
2.3 Les unités de capacité
2.4 Mémoire centrale : définitions
2.5 Mémoire centrale : caractéristiques

3. Une petite machine pédagogique 8 bits " PM "

3.1 Unité centrale de PM (pico-machine)
3.2 Mémoire centrale de PM
3.3 Jeu d'instructions de PM

1. Les principaux constituants

Un ordinateur, nous l'avons déjà noté, est composé d'un centre et d'une périphérie. Nous allons nous intéresser au cœur d'un ordinateur mono-processeur. Nous savons que celui-ci est composé de :

Une Unité centrale comportant :

- Unité de traitement,
- Unité de contrôle,
- Unités d'échanges.
- Une Mémoire Centrale.

1.1 Dans l'Unité Centrale : l'Unité de Traitement

Elle est chargée d'effectuer les traitements des opérations de types arithmétiques ou booléennes. L'UAL est son principal constituant.

Elle est composée au minimum :

- d'un registre de données **RD**
- d'un accumulateur **ACC**
- d'une unité arithmétique et logique : **UAL**

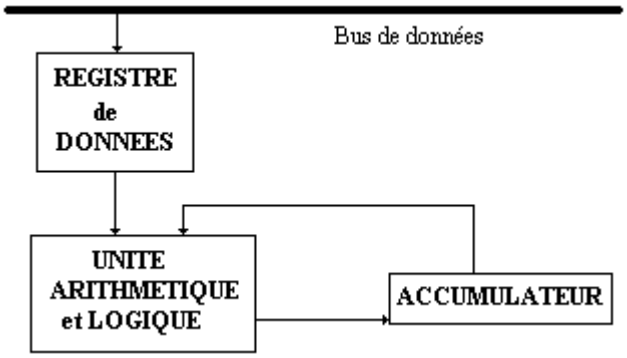


Schéma général de l'unité de traitement

La fonction du registre de données (mémoire rapide) est de contenir les données transitant entre l'unité de traitement et l'extérieur.

La fonction de l'accumulateur est principalement de contenir les opérandes ou les résultats des opérations de l'UAL.

La fonction de l'UAL est d'effectuer en binaire les traitements des opérations qui lui sont soumises et

qui sont au minimum:

- Opérations arithmétiques binaires: addition,multiplication, soustraction, division.
- Opérations booléennes : et, ou, non.
- Décalages dans un registre.

Le résultat de l'opération est mis dans l'accumulateur (Acc).

*1.2 Dans l'Unité Centrale : l'*Unité de Contrôle *ou de Commande*

Elle est chargée de commander et de gérer tous les différents constituants de l'ordinateur (contrôler les échanges, gérer l'enchaînement des différentes instructions, etc...)

Elle est composée au minimum de :

- d'un registre instruction **RI**,
- d'un compteur ordinal **CO**,
- d'un registre adresse **RA**,
- d'un décodeur de fonctions,
- d'une horloge.

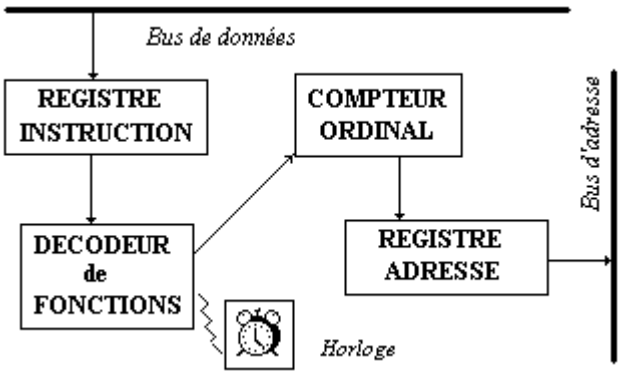


Schéma général de l'unité de contrôle

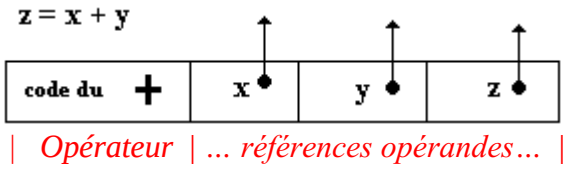
Vocabulaire **:**

Bit = plus petite unité d'information binaire (un objet physique ayant deux états représente un bit).

Processeur central = unité de commande + unité de traitement.

Instruction = une ligne de texte comportant un code opération, une ou plusieurs références aux opérandes.

Soit l'instruction fictive d'addition du contenu des deux mémoires x et y dont le résultat est mis dans une troisième mémoire z :



Registre instruction = contient l'instruction en cours d'exécution, elle demeure dans ce registre pendant toute la durée de son exécution.

Compteur ordinal = contient le moyen de calculer l'adresse de la prochaine instruction à exécuter.

Registre adresse = contient l'adresse de la prochaine instruction à exécuter.

Décodeur de fonction = associé au registre instruction, il analyse l'instruction à exécuter et entreprend les actions appropriées dans l'UAL ou dans la mémoire centrale.

1.3 Dans l'Unité Centrale : les Unités d'échange

- Une unité d'échange est spécialisée dans les entrées/sorties.
- Ce peut être un simple canal, un circuit ou bien un processeur particulier.
- Cet organe est placé entre la mémoire et un certain nombre de périphériques (dans un micro-ordinateur ce sont des cartes comme la carte son, la carte vidéo, etc...).

Une unité d'échange soulage le processeur central dans les tâches de gestion du transfert de l'information.

Les périphériques sont très lents par rapport à la vitesse du processeur (rapport de 1 à 10^9). Si le processeur central était chargé de gérer les échanges avec les périphériques il serait tellement ralenti qu'il passerait le plus clair de son temps à attendre.

1.4 Exemple *de machine à une adresse : un micro-processeur*

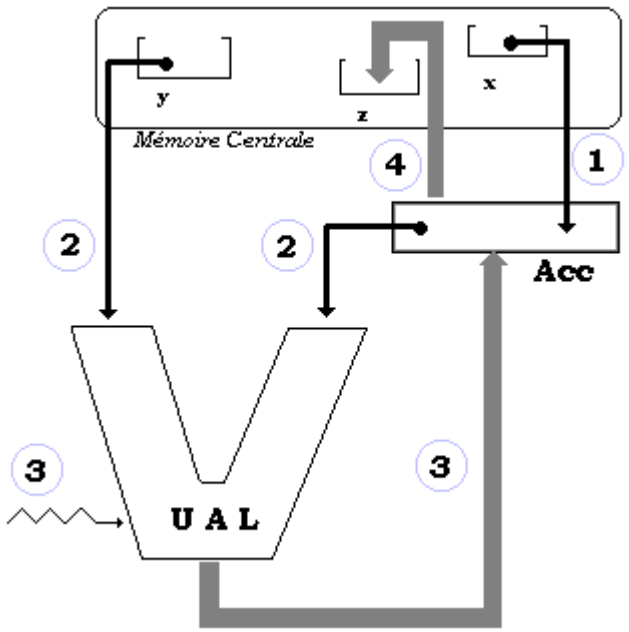
Un micro-processeur a les mêmes caractéristiques que celles d'un processeur central avec un niveau de complexité et de sophistication moindre.

Un micro-processeur est essentiellement une machine à une adresse, c'est à dire une partie code opérande et une référence à un seul opérande. Ce genre de machine est fondé sur un cycle de passage par l'accumulateur.

L'opération précédente z = x + y , se décompose dans une telle machine fictivement en 3 opérations distinctes illustrées par la figure ci-après :

LoadAcc x { chargement de l'accumulateur avec x : **(1)** }
Add y { préparation des opérandes x et y vers l'UAL : **(2)** }
{ lancement commande de l'opération dans l'UAL : **(3)** }
{ résultat transféré dans l'accumulateur : **(3)** }
Store z { copie de l'accumulateur dans z : **(4)** }

L'accumulateur gardant son contenu au final.



1.5 Les Bus

Un bus est un dispositif destiné à assurer le transfert simultané d'informations entre les divers composants d'un ordinateur.

On distingue trois catégories de Bus :

Bus d'adresses (unidirectionnel)

il permet à l'unité de commande de transmettre les adresses à rechercher et à stocker.

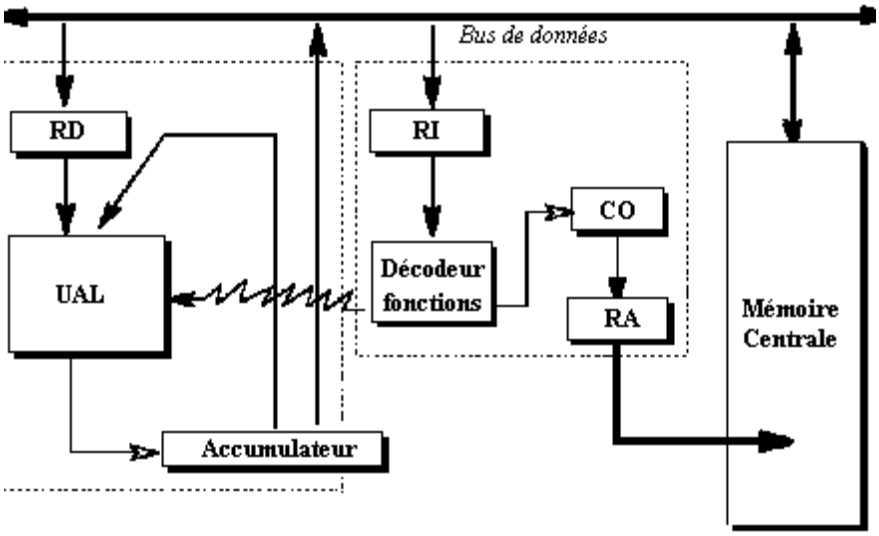
Bus de données (bi-directionnel)

sur lequel circulent les instructions ou les données à traiter ou déjà traitées en vue de leur rangement.

Bus de contrôle (bi-directionnel)

transporte les ordres et les signaux de synchronisation provenant de l'unité de commande vers les divers organes de la machine. Il véhicule aussi les divers signaux de réponse des composants.

1.6 Schéma général d'une micro-machine fictive



1.7 Notion de jeu d'instructions-machine : Les premiers programmes

Comme défini précédemment, une instruction-machine est une instruction qui est directement exécutable par le processeur.

L'ensemble de toutes les instructions-machine exécutables par le processeur s'appelle le " jeu d'instructions " de l'ordinateur. Il est composé au minimum de quatre grandes classes d'instructions dans les micro-processeurs :

- instructions de traitement
- instructions de branchement ou de déroutement
- instructions d'échanges
- instructions de comparaisons

D'autres classes peuvent être ajoutées pour améliorer les performances de la machine (instructions de gestion mémoire, multimédias etc..)

2. Mémoires - Mémoire Centrale

2.1 Mémoire

Mémoire :c'est un organe (électronique de nos jours), capable de contenir, de conserver et de restituer sans les modifier de grandes quantités d'information.

2.2 Les différents types de mémoires

La mémoire vive RAM (Random Access Memory)

- Mémoire dans laquelle on peut lire et écrire.
- Mémoire *volatile* (perd son contenu dès la coupure du courant).

La mémoire morte ROM (Read Only Memory)

- Mémoire dans laquelle on **ne** peut **que** lire.
- Mémoire *permanente* (conserve indéfiniment son contenu).

Les PROM (Programable ROM)

- Ce sont des mémoires vierges programmables une seule fois avec un outil spécialisé s'appelant un programmateur de PROM.
- Une fois programmées elles se comportent dans l'ordinateur comme des ROM.

Les EPROM (Erasable PROM)

- Ce sont des PROM effaçables (généralement sous rayonnement U.V),
- elles sont reprogrammables avec un outil spécialisé,
- elles se comportent comme des ROM en utilisation courante.

2.3 Les unités de capacité

Les unités de mesure de stockage de l'information sont :

Le bit (pas de notation)
L'octet = 2^3 bits = 8 bits. (noté 1 o)
Le Kilo-octet = 2^{10} octets =1024 o (noté 1 Ko)
Le Méga-octet = 2^{20} octets =$(1024)^2$ o (noté 1 Mo)
Le Giga-octet = 2^{30} octets =$(1024)^3$ o (noté 1 Go)
Le Téra-octet = 2^{40} octets =$(1024)^4$ o (noté 1 To)…

Les autres sur-unités sont encore peu employées actuellement.

2.4 Mémoire centrale : définitions

Mot : c'est un regroupement de **n** bits constituant une case mémoire dans la mémoire centrale. Ils sont tous numérotés.

Adresse : c'est le numéro d'un mot-mémoire (case mémoire) dans la mémoire centrale.

Programme : c'est un ensemble d'***instructions*** préalablement codées (en binaire) et enregistrées dans la mémoire centrale sous la forme d'une liste ***séquentielle*** d'instructions. Cette liste représente une suite d'actions élémentaires que l'ordinateur doit accomplir sur des ***données*** en entrée, afin d'atteindre le ***résultat*** recherché.

Organisation : La mémoire centrale est organisée en bits et en mots. Chaque mot-mémoire est repéré bijectivement par son **adresse** en mémoire centrale.

Contenu : La mémoire centrale contient en binaire, deux sortes d'informations

- des ***programmes***,
- des ***données***.

Composition : Il doit être possible de lire et d'écrire dans une mémoire centrale. Elle est donc habituellement composée de mémoires de type **RAM**.

Remarques

- Un ordinateur doté d'un programme est un automatisme apte seulement à répéter le même travail(celui dicté par le programme).
- Si l'on change le programme en mémoire centrale, on obtient un nouvel automatisme.

2.5 Mémoire centrale : caractéristiques

La mémoire centrale peut être réalisée grâce à des technologies différentes. Elle possède toujours des caractéristiques générales qui permettent de comparer ces technologies. En voici quelques unes :

La capacité représente le nombre maximal de mots que la mémoire peut stocker simultanément.
Le temps d'accès est le temps qui s'écoule entre le stockage de l'adresse du mot à sélectionner et l'obtention de la donnée.
Le temps de cycle ou cycle mémoire est égal au temps d'accès éventuellement additionné du temps de rafraîchissement ou de réécriture pour les mémoires qui nécessitent ces opérations.
La volatilité, la permanence.

Terminons ce survol des possibilités d'une mémoire centrale, en indiquant que le mécanisme d'accès à une mémoire centrale par le processeur est essentiellement de type séquentiel et se décrit selon trois phases :

- stockage,
- sélection,
- transfert.

Un ordinateur est une machine séquentielle de Von Neumann dans laquelle s'exécutent ces 3 phases d'une manière immuable, que ce soit pour les programmes ou pour les données.

3. Une petite machine pédagogique 8 bits

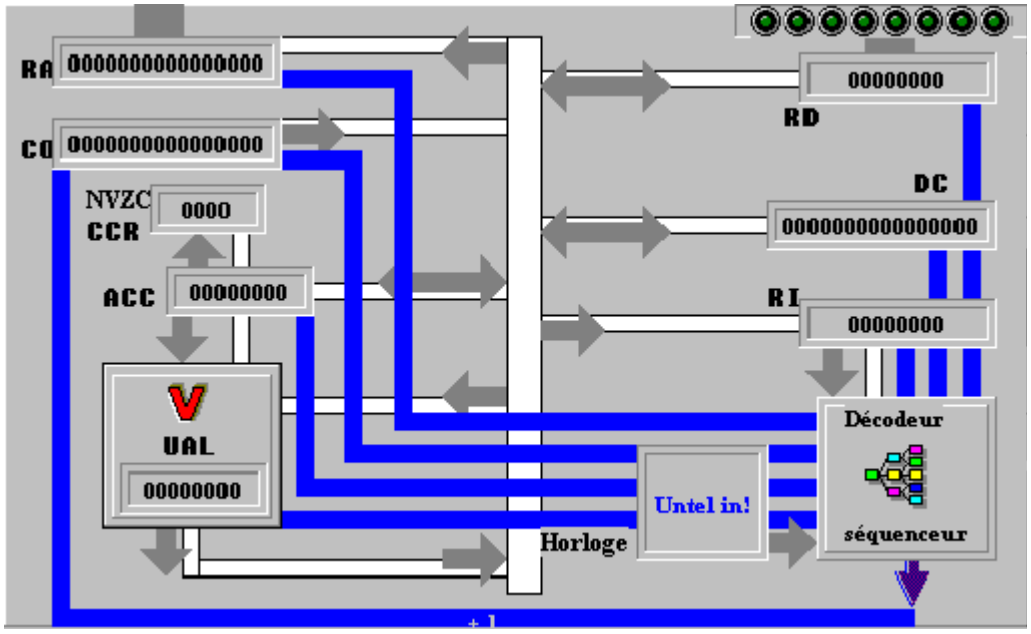
3.1 Unité centrale de PM (pico-machine)

Objectif:

Support pédagogique destiné à faire comprendre l'analyse et le cheminement des informations dans un processeur central d'ordinateur fictif.

- La mémoire centrale est à mots de 8 bits, les adresses sont sur 16 bits.
- le processeur est doté d'instructions immédiates ou relatives.
- Les instructions sont de 3 types à 1,2 ou 3 octets.

Description générale de l'unité centrale de PM simulée sur le tableau de bord ci-dessous :

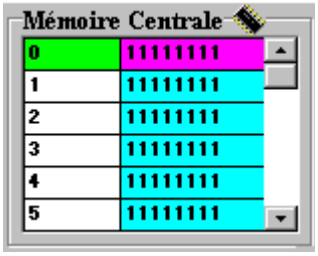


RA = Registre Adresse sur 16 bits
CO = Compteur Ordinal sur 16 bits
DC = Registre de formation d'adresse sur 16 bits
RD = Registre de Données sur 8 bits
UAL = Unité Arithmétique et Logique effectuant les calculs sur 8 bits avec possibilité de débordement.
Acc = Accumulateur sur 8 bits (machine à une adresse).
RI = Registre Instruction sur 8 bits (instruction en cours d'exécution).
Décodeur de fonction.

séquenceur
Horloge
CCR = un Registre de 4 Codes Condition **N, V, Z, C,**
BUS de contrôle (bi-directionnel)
BUS interne (circulation des informations internes).

3.2 Mémoire centrale de PM

La mémoire centrale de PM est de 512 octets, ce qui permet dans une machine 8 bits de voir comment est construite la technique d'adressage court (8 bits) et d'adressage long (16 bits).



|***adresse***|***contenu***|

Elle est connectée à l'unité centrale à travers deux bus : un bus d'adresse et un bus de données.

3.3 Jeu d'instructions de PM

PM est doté du jeu d'instructions suivant :

addition avec l'accumulateur
ADD #<valeur 2 octets code=16
ADD <adr 16 bits 3 octets code=18
ADD <adr 8 bits 2 octets code=17

chargement de l'accumulateur
LDA #<valeur 2 octets code=10
LDA <adr 16 bits 3 octets code=12
LDA <adr 8 bits 2 octets code=11

rangement de l'accumulateur

STA <adr 16 bits 3 octets code=15
STA <adr 8 bits 2 octets code=14

positionnement indicateurs CNVZ
STC (C=1) 1 octet code=100 **STN (N=1) 1 octet code=101** **STV (V=1) 1 octet code=102** **STZ (Z=1) 1 octet code=103** **CLC (C=0) 1 octet code=104** **CLN (N=0) 1 octet code=105** **CLV (V=0) 1 octet code=106** **CLZ (Z=0) 1 octet code=107**

branchement relatif sur indicateur
BCZ (brancht.si C=0) 2 octets code=22 **BNZ (brancht.si N=0) 2 octets code=23** **BVZ (brancht.si V=0) 2 octets code=24** **BZZ (brancht.si Z=0) 2 octets code=25** **END (fin programme) 1 octet code=255**

Dans le CCR les 4 bits indicateurs sont dans cet ordre : N V Z C.
Ils peuvent être :

- soit positionnés automatiquement par la machine:

N = le bit de poids fort de l'Accumulateur
V = 1 si overflow (dépassement capacité) 0 sinon
Z = 1 si Accumulateur vaut 0
Z = 0 si Accumulateur <0
C = 1 si retenue (dans l'addition) sinon 0

- soit positionnés par programme.

Exemple de programme en PM
LDA #18 ; *{chargement de l'accumulateur avec la valeur 18}* **STA 50 ;** *{rangement de l'accumulateur dans la mémoire n° 50}* **LDA #5 ;** *{chargement de l'accumulateur avec la valeur 5}* **STA 51 ;** *{rangement de l'accumulateur dans la mémoire n°51}* **ADD 50 ;** *{addition de l'accumulateur avec la mémoire n°50}* **STA 52 ;** *{rangement de l'accumulateur dans la mémoire n°52}* **END**

1.6 Système d'exploitation

Plan du chapitre:

1. Notion de système d'exploitation

1.1 Les principaux types d' OS

monoprogrammation

multi-programmation

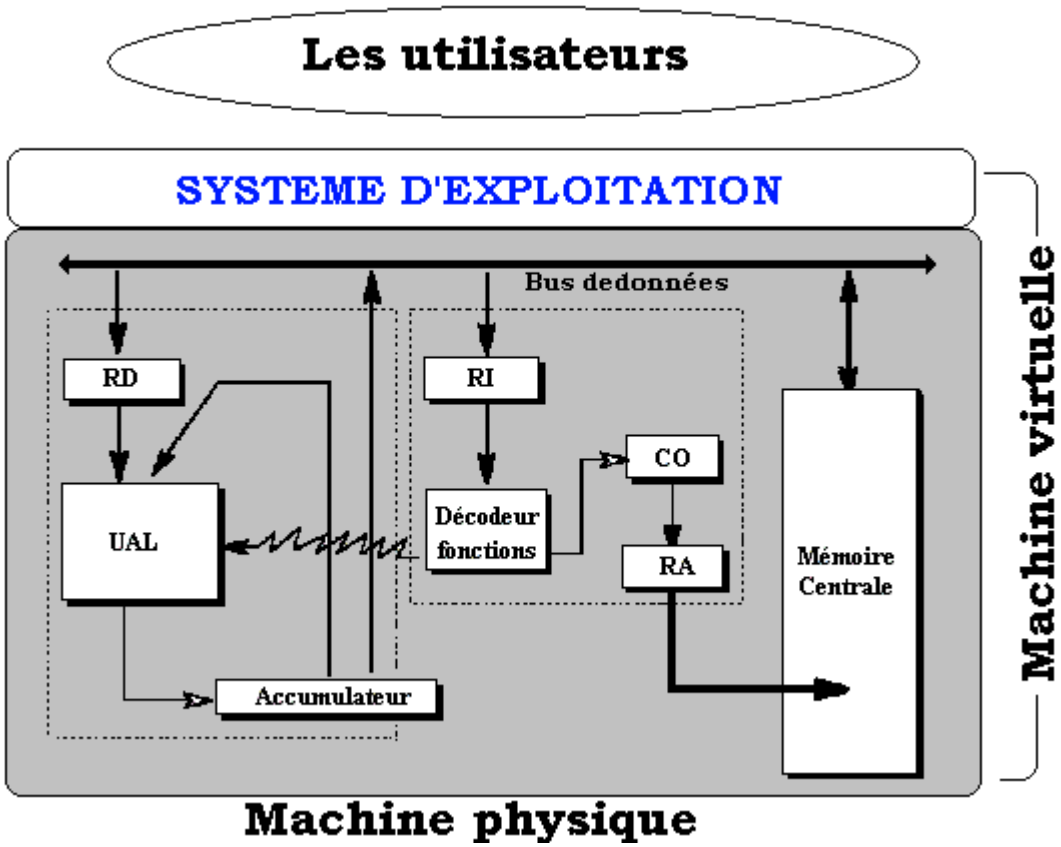
temps partagé

2. Les OS des micro-ordinateurs

2.1 Le système d'exploitation Windows de Microsoft

1. Notion de système d'exploitation

Un ordinateur est constitué de **matériel** (hardware) et de **logiciel** (software). Cet ensemble est à la disposition de un ou plusieurs utilisateurs. Il est donc nécessaire que quelque chose dans l'ordinateur permette la communication entre l'homme et la machine. Cette entité doit assurer une grande souplesse dans l'interface et doit permettre d'accéder à toutes les fonctionnalités de la machine. Cette entité douée d'une certaine intelligence de communication se dénomme " la machine virtuelle ". Elle est la réunion du matériel et du système d'exploitation (que nous noterons OS par la suite).



Le système d'exploitation d'un ordinateur est chargé d'assurer les fonctionnalités de communication et d'interface avec l'utilisateur. Un OS est un logiciel dont le grand domaines d'intervention est la gestion de toutes les ressources de l'ordinateur :

- mémoires,
- fichiers,
- périphériques,
- entrée-sortie,
- interruptions, synchronisation...

Un système d'exploitation n'est pas un logiciel unique mais plutôt une famille de logiciels. Une partie de ces logiciels réside en mémoire centrale (nommée **résident** ou **superviseur**), le reste est stocké en mémoire de masse (disques durs par exemple).

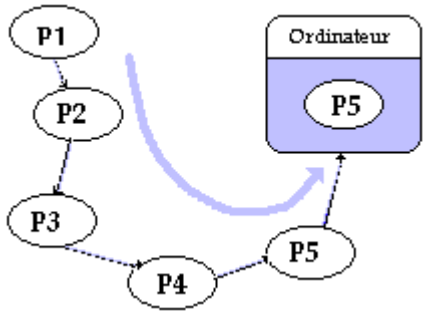
Afin d'assurer une bonne liaison entre les divers logiciels de cette famille, la cohérence de l'OS est généralement organisée à travers des tables d'interfaces architecturées en couches de programmation (niveaux abstraits de liaison). La principale tâche du superviseur est de gérer le contrôle des échanges d'informations entre les diverses couches de l'OS.

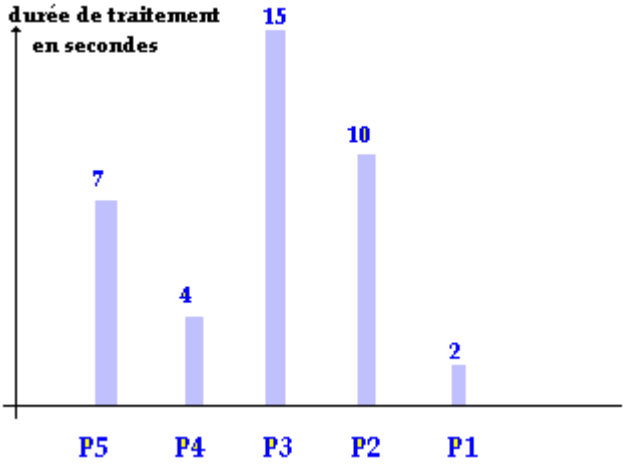
1.1 Les principaux types d' OS

Nous avons vu dans le tableau synoptique des différentes générations d'ordinateurs que les OS ont subi une évolution parallèle à celle des architectures matérielles. Nous observons en première approximation qu'il existe trois types d'OS différents, si l'on ignore les systèmes rudimentaires de la 1ère génération.

MONOPROGRAMMATION : La 2ème génération d'ordinateurs est équipée d'OS dits de " monoprogrammation " dans lesquels un seul utilisateur est présent et a accès à toutes les ressources de la machine pendant tout le temps que dure son travail. L'OS ne permet le passage que d'un seul programme à la fois.

A titre d'exemple, supposons que sur un tel système 5 utilisateurs exécutent chacun un programme P_1, P_2, P_3, P_4, P_5 :





Dans l'ordre de la figure ci-haut, chaque P_i attend que le P_{i+1} précédent ait terminé son exécution pour être exécuté à son tour.

Exemple de diagramme des temps d'exécution de chaque programme P_i de la figure de gauche.

L'axe des abscisses du diagramme des temps d'exécution, indique l'ordre de passage précédent (P_5, puis P_4 etc...) nous voyons que les temps d'attente d'un utilisateur ne dépendent pratiquement pas de la durée d'exécution de son programme mais surtout de l'ordre du passage (les derniers sont pénalisés surtout si en plus leur temps propre d'exécution est faible comme P_1 par exemple).

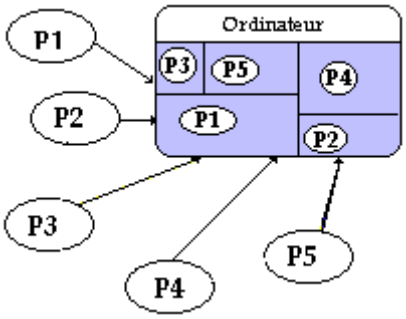
Une vision abstraite et synthétique d'un tel système est de considérer que 5 tables suffisent à le décrire. La table :

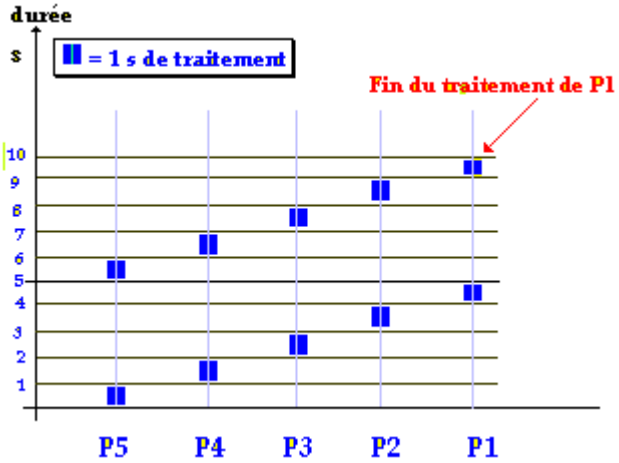
- des unités logiques,
- des unités physiques,
- des états,
- de ventilation des interruptions,
- des canaux.

Relativement aux temps d'attente, un système de monoprogrammation est injuste vis à vis des petits programmes.

MULTIPROGRAMMATION : La 3ème génération d'ordinateur a vu naître avec elle les OS de multiprogrammation. Dans un tel système, plusieurs utilisateurs peuvent être présents en " même temps " dans la machine et se partagent les ressources de la machine pendant tout leur temps d'exécution.

En reprenant le même exemple que précédemment, P_1, P_2, P_3, P_4, P_5 sont exécutés cycliquement par l'OS qui leur alloue les ressources nécessaires (disque, mémoire, fichier,...) pendant leur tranche de temps d'exécution. Nous exposons dans l'exemple ci-dessous uniquement des exécutions ne nécessitant jamais d'interruptions, ni de priorité, et nous posons comme hypothèse que le temps fictif alloué pour l'exécution est de 1 seconde :





Dans la figure ci-haut, chaque P_i se voit allouer une tranche de temps d'exécution (1 seconde), dès que ce temps est écoulé, l'OS passe à l'exécution du P_{i+1} suivant etc…

Exemple de diagramme des temps d'exécution cyclique de chaque programme P_i de la figure de gauche.

Nous observons dans le diagramme des temps d'exécution que le système exécute P_5 pendant 1 seconde, puis abandonne P_5 et exécute P_4 pendant 1 seconde, puis abandonne P_4..., jusqu'à l'exécution de P_1, lorsqu'il a fini le temps alloué à P_1, il recommence à parcourir cycliquement la liste (P_5, P_4, P_3, P_2, P_1)et réalloue 1 seconde de temps d'exécution à P_5 etc... jusqu'à ce qu'un programme ait terminé son exécution et qu'il soit sorti de *la table des programmes à exécuter*.

Une vision abstraite déduite du paragraphe précédent et donc simplificatrice, est de décrire un tel système comme composé des 5 types de tables précédentes en y rajoutant de nouvelles tables et en y incluant la notion de priorité d'exécution hiérarchisée. Les programmes se voient affecter une priorité qui permettra à l'OS selon les niveaux de priorité, de traiter certains programmes plus complètement ou plus souvent que d'autres.

Relativement aux temps d'attente, un système de multiprogrammation rétablit une certaine justice entre petits et gros programmes.

TEMPS-PARTAGE : Il s'agit d'une amélioration de la multiprogrammation orientée vers le transactionnel. Un tel système organise ses tables d'utilisateurs sous forme de files d'attente. L'objectif majeur est de connecter des utilisateurs directement sur la machine et donc d'optimiser les temps d'attente de l'OS (un humain étant des millions de fois plus lent que la machine sur ses temps de réponse).

La 4ème génération d'ordinateur a vu naître les réseaux d'ordinateurs connectés entre eux et donc de nouvelles fonctionnalités, comme l'interfaçage réseau, qui ont enrichi les OS déjà existants. De nouveaux OS entièrement orientés réseaux sont construits de nos jours.

2. Les OS des micro-ordinateurs

Les micro-ordinateurs apparus dans le grand public dès 1978 avec le Pet de Commodore, l'Apple et l'IBM-PC, ont répété en accéléré les différentes phases d'évolution des générations d'ordinateurs. Les OS des micro-ordinateurs ont suivi la même démarche et sont partis de systèmes de monoprogrammation comme MS-DOS et MacOS pour évoluer en systèmes multi-tâches (version affaiblie de la multiprogrammation) avec OS/2 , windows et Linux.

De nos jours un OS de micro-ordinateur doit nécessairement adopter des normes de convivialité dans la communication homme-machine sous peine d'être rejeté par le grand public. Là, gît à notre sens, un des seuls intérêts de l'impact puissant du marché sur l'informatique. La pression des masses de consommateurs a fait sortir l'informatique des milieux d'initiés, et s'il n'y avait pas cette pression, les OS seraient encore accessibles uniquement par des langages de commandes textuels dont les initiés raffolent (la compréhension d'un symbolisme abstrus dénotant pour certains la marque d'une supériorité toute illusoire et assez insignifiante). Notons aussi que la réticence au changement, la résistance à la nouveauté et la force de l'habitude sont des caractéristiques humaines qui n'ont pas favorisé le développement des interfaces de communication. La communication conviviale des années 90-2000 réside essentiellement dans des notions inventées dans les années 70-80 à Xerox PARC (Palo Alto Research Center of Xerox), comme la souris, les fenêtres, les menus déroulants, les icônes, et que la firme Apple a commercialisé la première dans l'OS du MacIntosh dès 1984.

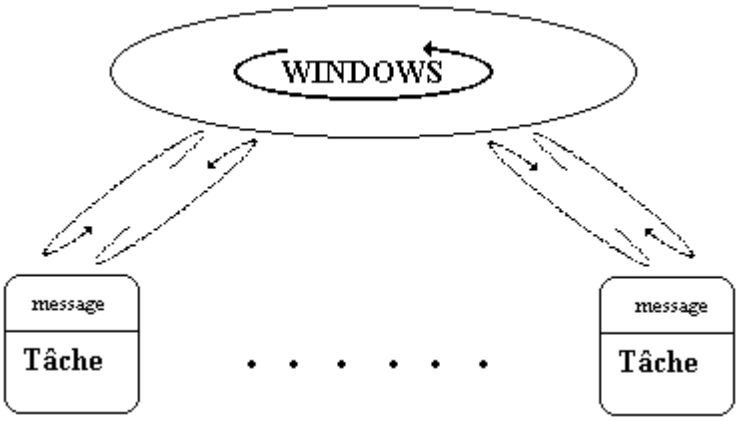
Windows de Microsoft et OS/2 d'IBM se sont d'ailleurs ralliés à cette ergonomie.

LINUX fondé sur **UNIX** et en distribution officiellement "gratuite" sous licence GNU, essaie de concurrencer le système Windows sur PC, le match est encore inégal en nombre de logiciels installés fonctionnant sous cet OS, malgré un important battage médiatique effectué autour de ce système dans la fin des années 90 et les rumeurs récurrentes de la disparition de Windows voir même de la société Microsoft.

Le BeOs est un autre système Unix-like développé pour micro-ordinateur lui aussi fondé sur des logiciels GNU mais il est officiellement payant (le prix est modeste et équivalent aux distributions de Linux).

2.1 Le système d'exploitation Windows de Microsoft

Un micro-ordinateur est constitué de matériel (hardware) que l'on trouve partout dans le commerce (y compris dans les grandes surfaces), et de logiciels fonctionnant sous un système d'exploitation. Le marché éducatif français est encore occupé aujourd'hui à plus de 90% par le système Windows de la société Microsoft.



WINDOWS 3.1 était un système coopératif utilisant le time-slicing.
Depuis WINDOWS 95, les successeurs de Windows sont des systèmes préemptifs.

Pour un système, le qualificatif coopératif s'oppose à préemptif.

Environnement coopératif : c'est un environnement dans lequel les applications doivent régulièrement donner le contrôle au noyau du système (KERNEL.EXE dans Windows) afin que celui-ci puisse répartir le temps du processeur aux autres applications.

Si une application windows 3.1 est appelée (exécutée) et qu'elle "garde la main" (c'est à dire ne retourne pas à Windows), toutes les autres applications (ou tâches) exécutées "en même temps" ne peuvent alors, ni être appelées, ni être exécutées et restent donc bloquées.

Noyau du système :

Il fait partie du résident et s'exécute dans un mode non interruptible. Il gère entre autres activités l'ordonnancement des tâches, il assure la gestion de la mémoire centrale, il effectue le partage des ressources.

Time-slicing :

C'est une méthode de répartition du temps (utilisée par windows par exemple) entre différentes tâches. Le noyau découpe le temps d'utilisation en tranches (unités minimales de temps), lesquelles sont allouées aux tâches en cours d'exécution (cette allocation se fait en fonction de leur priorité, le noyau allouant ainsi plus ou moins de tranches selon le niveau de priorité).

Environnement préemptif : C'est un environnement semblable à un environnement coopératif dans lequel le noyau du système peut interrompre à tout moment une tâche quelconque pour donner le contrôle à une autre tâche (ou application).

Rappelons que si Windows (Version 3.x) n'était pas un vrai système d'exploitation multi-tâche, ses successeurs (95, 98, Me, 2000, Xp,...) sont de vrais multi-tâche préemptifs et sont devenus de véritables systèmes de multiprogrammation complets en particulier la dernière version baptisée LongHorn contenant le FrameWork .Net.

1.7 Réseaux

Plan du chapitre:

1. Les réseaux d'ordinateurs

1.1 Les différentes topologies de réseaux
1.2 Réseau local

2. Liaisons entre réseaux

2.1 Réseau à commutation de paquets
2.2 Le réseau mondial Internet
2.3 Intranet

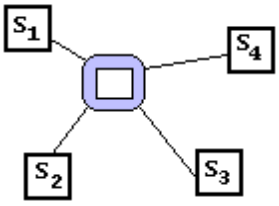
1. Les réseaux d'ordinateurs

L'objectif principal d'un réseau d'ordinateurs est de relier et de permettre l'exploitation à distance de systèmes informatiques à l'aide des télécommunications dans le cadre de réseaux à grande distance (les réseaux locaux emploient une technologie de câblage interne à l'entreprise). Il existe différentes manières d'interconnecter des systèmes informatiques à distance. On les nomme topologies de réseaux.

1.1 Les différentes topologies de réseaux

A) Le point à point simple

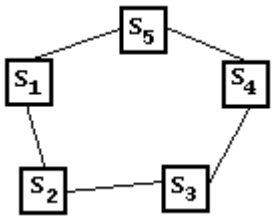
- n liaisons pour n systèmes $\mathbf{S_i}$ interconnectés,
- 1 seul point de connexion.



Architecture étoile

B)Le point à point en boucle

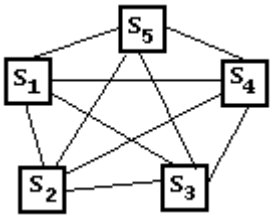
- n liaisons pour n systèmes $\mathbf{S_i}$ interconnectés,
- Chaque $\mathbf{S_i}$ passe l'information au $\mathbf{S_i}$ suivant.



Architecture anneau

C) Le point à point complet

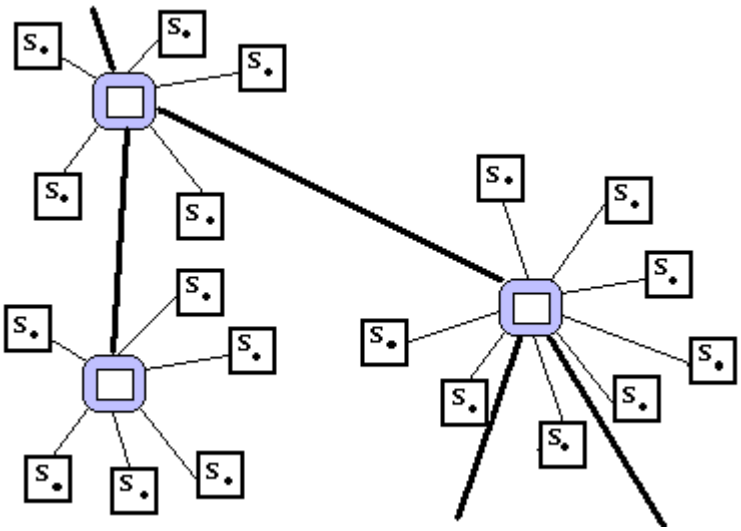
- n(n-1)/2 liaisons pour n systèmes $\mathbf{S_i}$ interconnectés,
- tous les $\mathbf{S_i}$ sont reliés entre eux.



Architecture maillée

D) Le point à point arborescent

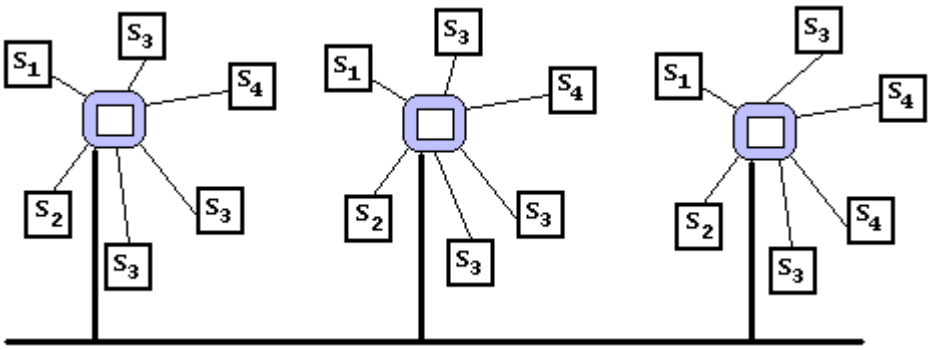
- n liaisons pour n systèmes S_i interconnectés à un même noeud,
- 1 liaison pour chaque noeud vers ses descendants,(topologie étoile à chaque noeud).



Architecture hiérarchique

E) Le multipoint

- n liaisons pour n systèmes S_i interconnectés à un même noeud,
- les points de connexion sont reliés par une même voie.



Architecture en bus

Il existe aussi des réseaux construits selon des combinaisons de ces topologies entre elles.

1.2 Réseau local

C'est un réseau dont les distances de liaison sont très faibles(entreprise, établissement scolaire, une salle,...).

Les réseaux locaux peuvent comporter ou non des ***serveurs*** (système informatique assurant la répartition et la gestion de ressources communes aux utilisateurs) et utiliser l'une des cinq architectures précédentes.

Ils sont composés de liaisons hertziennes ou établies par câble. Lorsqu'il y a plusieurs serveurs, chaque serveur peut être un poste de travail comme les autres ou bien être un ***serveur dédié*** (ne faisant office que de serveur). Signalons que la partie réseau local des micro-ordinateurs dotés d'un OS comme windows ne nécessite aucun serveur dédié mais fonctionne aussi avec une version du système de type serveur.

Les deux principaux standards qui se partagent l'essentiel du marché des réseaux locaux sont Ethernet (topologie en bus) et token-ring (topologie en anneau). Les protocoles (loi d'échange d'information entre les systèmes informatiques) sont très nombreux. Le plus utilisé quantitativement dans le monde est TCP/IP (Transfert Control Protocol/Internet Protocol) qui est un protocole synchrone orienté bit (les informations sont des suites de bits).

2. Liaisons entre réseaux d'ordinateurs

Il existe diverses techniques d'interconnexion de réseaux entre eux. Nous renvoyons le lecteur à des ouvrages spécialisés sur les réseaux. Nous allons brosser un tableau simple et général du réseau mondial le plus connu de nos jours au niveau du grand public, le réseau Internet. Nous verrons ensuite comment il est adapté par les spécialistes à des architectures locales sous la forme d'Intranet. En premier lieu donnons quelques explications techniques sur un mode classique de transmission de l'information utilisé par de nombreux réseaux (Internet, Transpac,...).

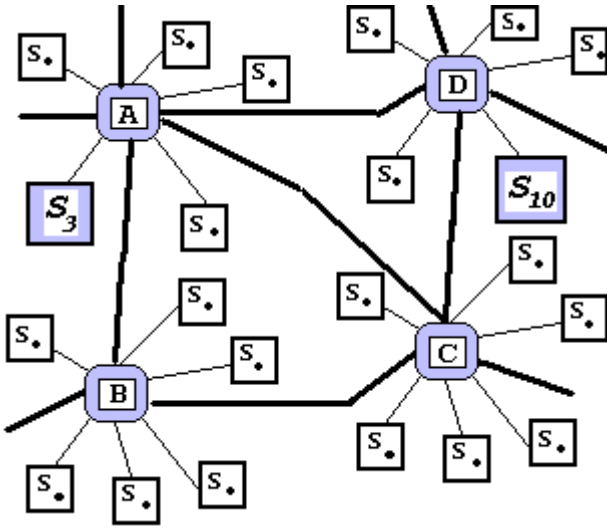
2.1 Réseau à commutation de paquets

Dans un tel type de réseau nous avons besoin de définir au moins **trois** concepts :

- **le message** : l'information échangée entre deux systèmes informatiques.
- **les paquets** : des petites suites de bits constituant une partie d'un message, (le message est découpé en plusieurs paquets).
- **le routage** : c'est l'action (effectuée par le routeur) qui permet la transmission, l'aiguillage et la redirection des informations circulant sur le réseau à un instant donné.

Un tel réseau est architecturé selon une topologie plus ou moins fortement maillée, entre les divers concentrateurs. Les utilisateurs $\mathbf{S_i}$ se connectent selon leur proximité géographique au concentrateur le plus proche.

Dans le schéma suivant, représentant une maille du réseau, nous supposons que l'utilisateur $\mathbf{S_3}$ veuille envoyer un message *M*(image, fichier, son, etc...) à $\mathbf{S_{10}}$**.** Nous allons suivre le chemin parcouru par les paquets $\mathbf{p_i}$ du message *M* pour aller de $\mathbf{S_3}$ à $\mathbf{S_{10}}$.



$\mathbf{S_3}$ est directement connecté au concentrateur [A], $\mathbf{S_{10}}$ est directement connecté au concentrateur [D]. Supposons aussi que le message *M* soit composé de 4 paquets : $M = (\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}, \mathbf{p_4})$.

Le routage de départ s'effectue à partir du concentrateur [A] et de la charge et de l'encombrement actuels du réseau. Ce sont ces deux critères qui permettent au routeur de prendre la décision d'émission des paquets.

Principe du routage :
Les paquets dans un tel réseau sont envoyés dans n'importe quel ordre et indépendamment les uns des autres vers des destinations diverses ; chaque paquet voyage bien sûr, avec l'adresse du destinataire du message.

a) Supposons que $\mathbf{p_1}$ aille directement vers [D], puis que l'encombrement oblige d'envoyer $\mathbf{p_2}$ à [B] puis $\mathbf{p_3}$, $\mathbf{p_4}$ à [C].

b) Puis [C] peut router directement $\mathbf{p_3}$, $\mathbf{p_4}$ vers [D] (qui a déjà reçu $\mathbf{p_1}$).

c) Enfin [B] envoie $\mathbf{p_2}$à [C] et celui-ci le redirige vers [D] (qui avait déjà reçu $\mathbf{p_1}$,$\mathbf{p_3}$ et $\mathbf{p_4}$).

d) Lorsque $\mathbf{p_2}$ arrive au concentrateur [D], le message *M* est complet, il peut être reconstitué $M = (\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}, \mathbf{p_4})$et expédié à son destinataire $\mathbf{S_{10}}$.

Internet

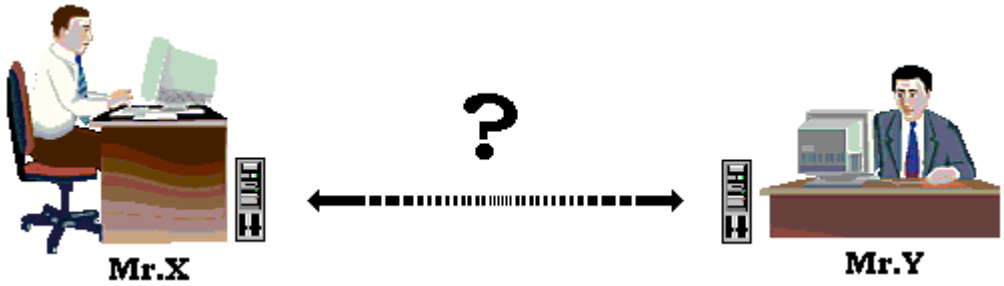
Le réseau le plus connu se dénomme **Internet**. Chaque pays peut avoir mis en place un réseau national, (par exemple en France, il existe un réseau national public TRANSPAC fonctionnant par commutations de paquets sous protocole X25), le réseau Internet quant à lui est international et fonctionne par commutations de paquets sous protocole TCP/IP.

C'est actuellement le réseau mondial de transmission de données le plus utilisé avec plusieurs centaines de millions d'utilisateurs.

- C'est un réseau à commutation de paquets.
- Il est basé sur le protocole TCP/IP.
- Il permet à des milliers d'autres réseaux locaux ou non de se connecter entre eux à distance.

Explication pratique de la transmission de données sur Internet

Prenons un exemple pratique, Mr. X situé à Moscou désire envoyer le message suivant "*Bonjour cher ami comment allez-vous ?*" à Mr. Y situé à Ankara, via le réseau Internet.



Protocole, adresse IP

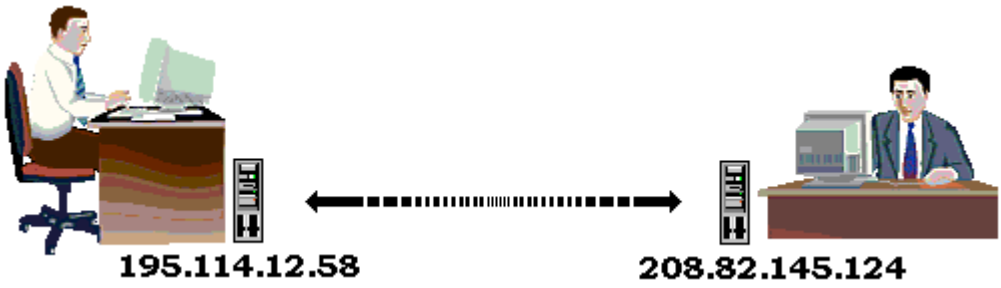
La communication entre deux machines distantes implique une normalisation des échanges sous forme de règles. Un tel ensemble de règles est appelé un **protocole de communication**. Un protocole décompose la communication en sous-problèmes simples à traiter dénommé **couche** du protocole. Chaque couche a une fonction précise et fait abstraction du fonctionnement des couches supérieures et inférieures.

Le protocole de communication TCP/IP utilisé par Internet, est composé de 4 couches: **application**, **transport**, **réseau** et **interface**

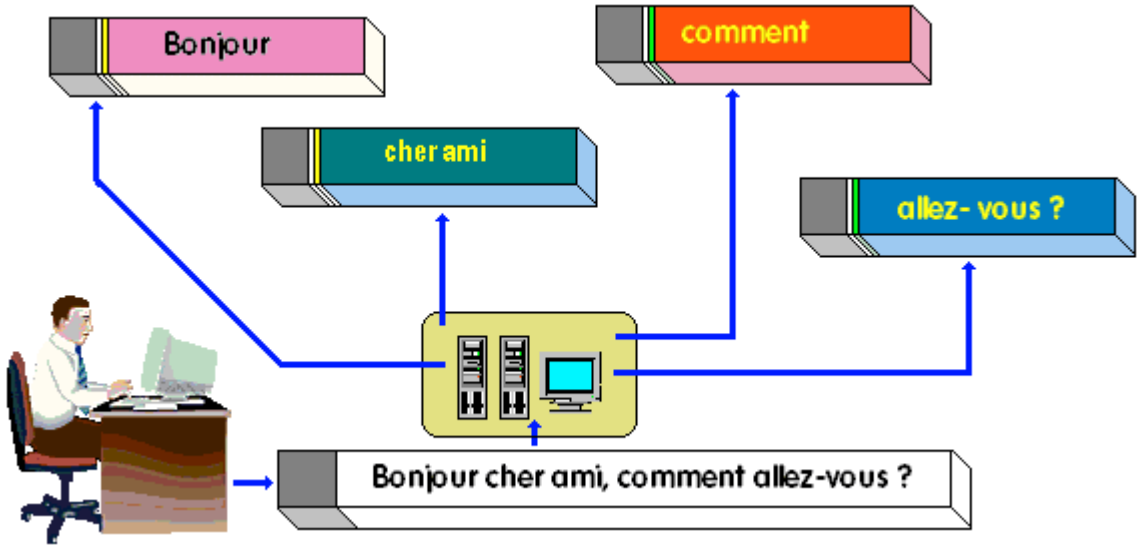
Un individu est identifiable par son numéro de sécurité sociale (deux personnes différentes n'ont pas le même numéro de sécurité sociale), de même chaque ordinateur branché sur Internet se voit attribuer un numéro unique qui permet de l'identifier.

On dénomme **adresse IP** un tel identifiant.Une adresse IP se présente sous la forme de 4 nombres (entre 0 et 255) que l'on sépare par des points pour des raisons de lisibilité , exemple : **163.85.210.8**.

Donc l'ordinateur de Mr. X situé à Moscou est connecté à Internet possède une adresse IP (par exemple : **195.114.12.58**), celui de Mr.Y possède aussi une adresse IP (par exemple : **208.82.145.124**)



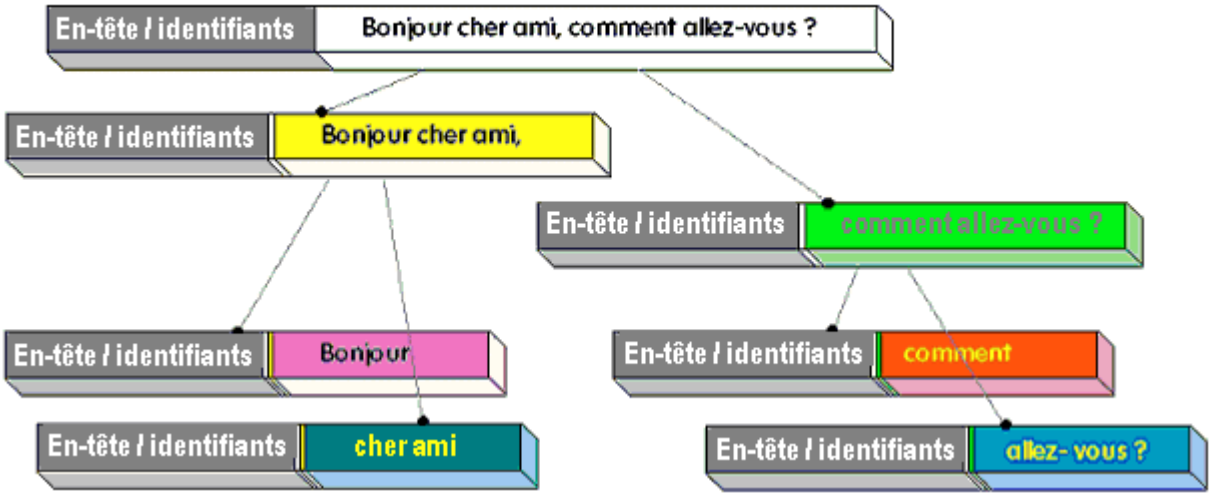
Le message initial de MrX va être découpé par TCP/IP, fictivement pour les besoins de l'exemple en quatre paquets (en fait la taille réelle d'un paquet IP est d'environ 1500 octets) :



La **Donnée** initiale de chacun des 4 paquets ("*Bonjour", cher ami" , "comment", "allez-vous ?")* est modifiée par chaque couche du protocole TCP/IP par l'ajout d'une **En-tête** spécifique nécessaire à la réalisation de la fonction de cette couche.

Plusieurs protocoles plus généraux sont fondés sur TCP/IP : SMTP,FTP, POP3, HTTP. Dans le cas d'HTTP, le paquet contient une partie identifiant supplémentaire :

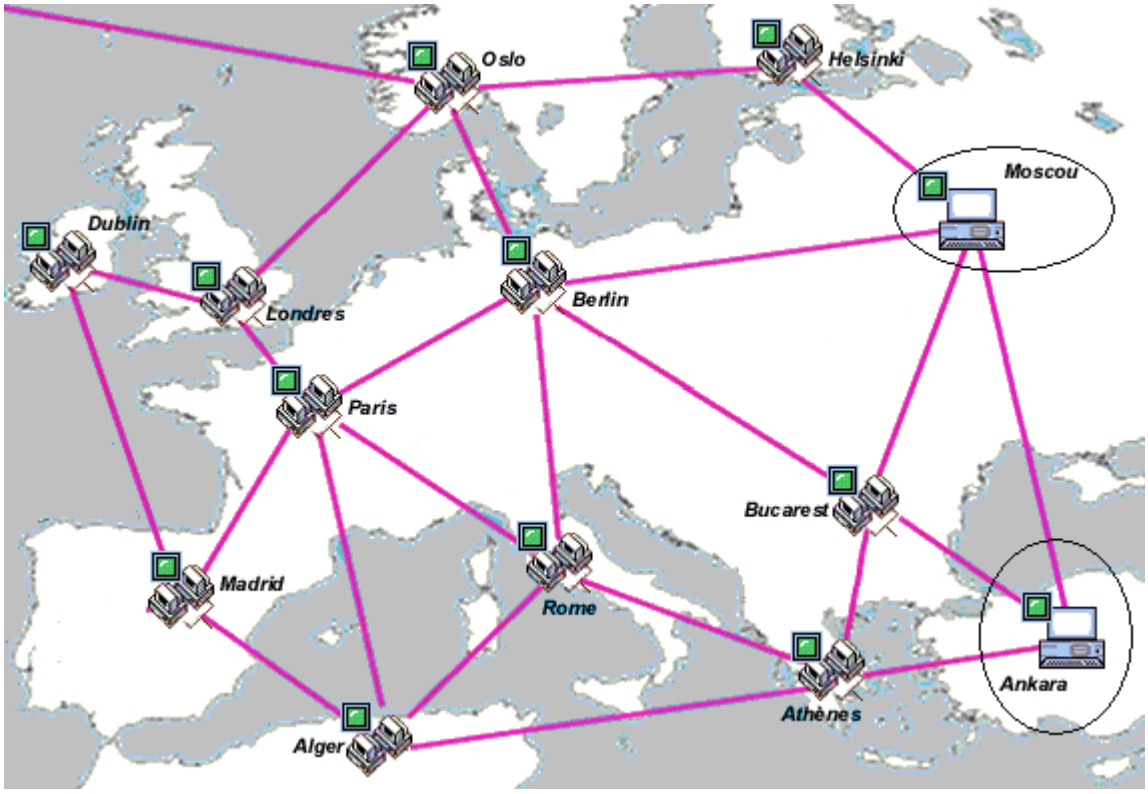
Le message initial de MrX est donc découpé avec les en-têtes adéquates **:**



(chaque en-tête/identifiant de paquet contient l'adresse de l'ordinateur de l'expéditeur MrX soit : **195.114.12.58** et celle du destinataire Mr.Y soit : **208.82.145.124**)

Le routage

Supposons que nous avons la configuration de connexion figurée ci-après :



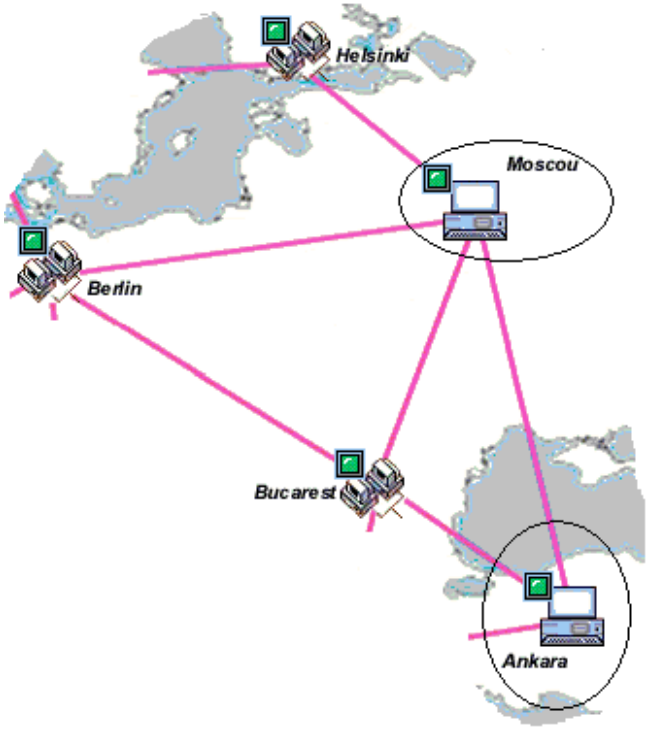
Le schéma précédent représente les points de routage fictifs du réseau Internet au voisinage de Moscou et Ankara.

Le routage sur Internet est l'opération qui consiste à trouver **le chemin le plus court** entre deux points du réseau en fonction en particulier de l'**encombrement** et de l'**état** du réseau.

Cette opération est effectuée par un routeur qui peut être soit un matériel spécifique raccordé à un ordinateur, soit un ordinateur équipé d'un logiciel de routage.

Chaque routeur dispose d'une **table** l'informant sur l'état du réseau, sur le routeur suivant en fonction de la destination et sur le nombre de routeurs nécessaires pour aller vers la destination.

Dans notre exemple, nous avons supposé que le routeur de Moscou soit branché avec les quatre routeurs d'**Ankara** , d'**Helsinki** , de **Berlin** et de **Bucarest** :



Informations collectées au moment de l'envoi du 1er paquet à partir de Moscou :

Ankara (état : en réparation)

Helsinki (état : disponible)

Berlin (état : disponible)

Bucarest (état : saturé)

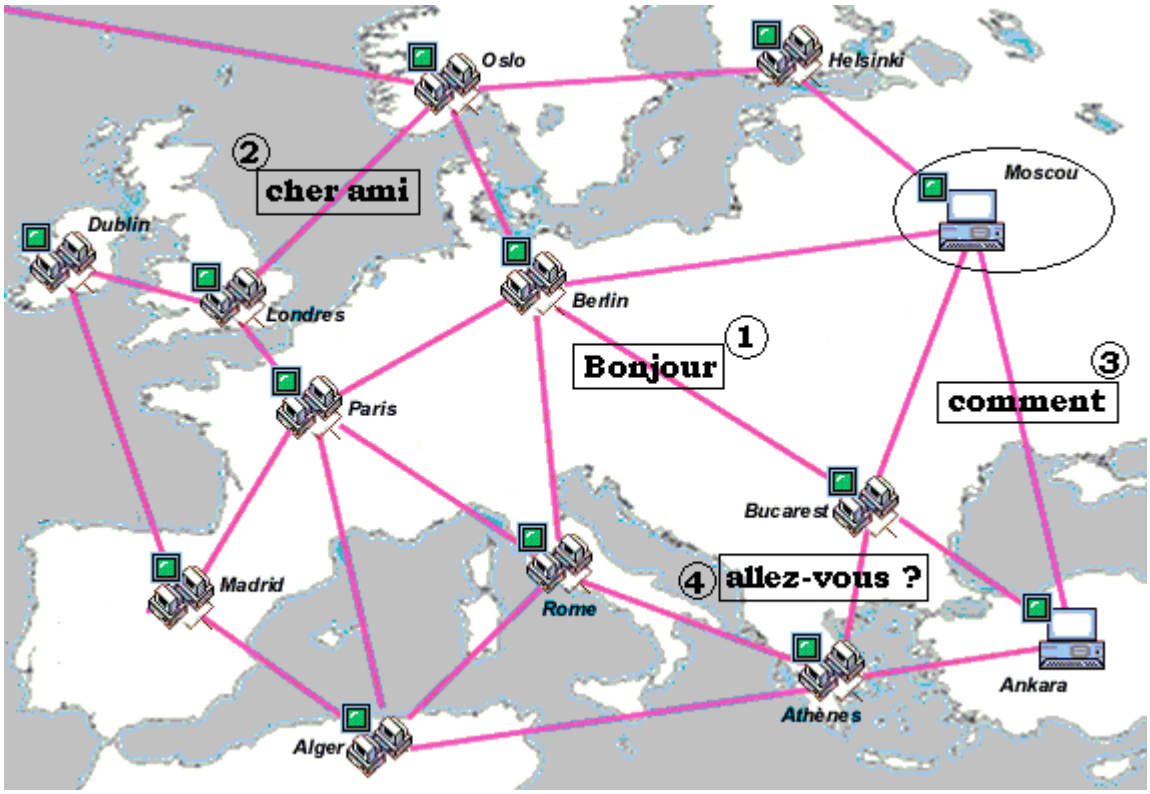
La table de routage aura à peu près cette allure :

Routeur	Destination	Nombre de routeurs	Routeur suivant	Etat
Moscou	Ankara	5	Helsinki	libre
Moscou	Ankara	2	Bucarest	saturé
Moscou	Ankara	3	Berlin	libre
Moscou	Ankara	1	Ankara	indisponible

Il est évident que d'après la table précédente seules deux destinations immédiates sont libres : le routeur d'Helsinki ou le routeur de Berlin.

Comme le nombre de routeurs restant à parcourir est moindre en direction de Berlin vers Ankara (3 routeurs : Berlin-Bucarest-Ankara) comparé à celui de la direction Helsinki vers Ankara (5 routeurs : Helsinki-Oslo-Berlin-Bucarest-Ankara), c'est le trajet Berlin qui est choisi pour le premier paquet "*Bonjour*".

Au bout de quelques instants, les 4 paquets obtenus à partir du message de Mr.X voyagent sur **Internet** indépendamment les uns des autres vers des destinations diverses (n'oublions pas que chaque paquet voyage avec l'adresse du destinataire du message qui est située à **Ankara**).



Carte : Le voyage des paquets

Sur cette carte :

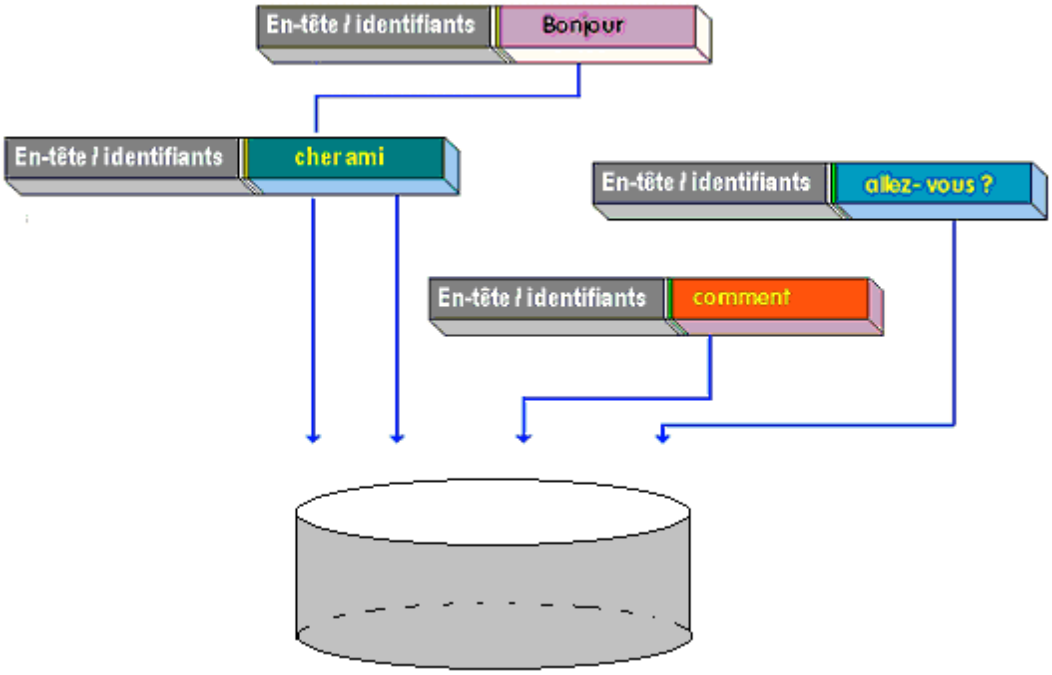
Le paquet n°1 *"Bonjour"* , voyage vers le routeur de Bucarest.

Le paquet n°2 *"cher ami"* , voyage vers le routeur de Londres.

Le paquet n°3 *"comment"* , voyage vers le routeur d'Ankara.

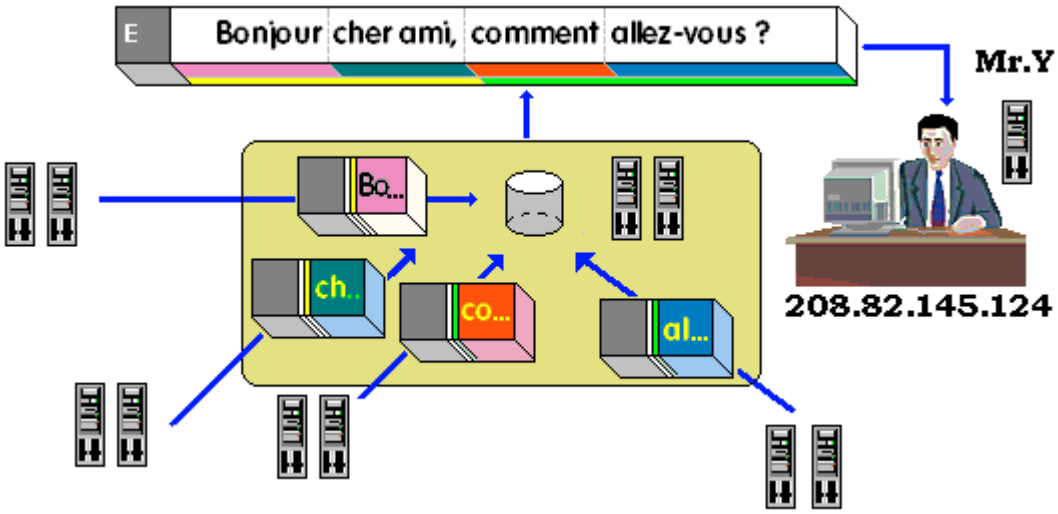
Le paquet n°4 *"allez-vous ?"* , voyage vers le routeur d'Athènes.

A l'arrivée à Ankara, le routeur d'Ankara reçoit les paquets en ordre dispersés et en des temps différents et les stocke en attendant que le message soit complet :



Le routeur d'Ankara vérifie que les paquets sont tous bien arrivés, il redemande éventuellement les paquets manquants, il envoi un accusé de réception pour prévenir chaque routeur expéditeur que les données sont bien arrivées.

Au final il y a réassemblage des paquets pour reconstituer le message original avant de le distribuer au logiciel de lecture du message de Mr.Y :



La petite histoire d'Internet

Le concept d'Internet n'est pas récent. Il prend naissance en effet à la fin des années soixante dans les rangs des services **militaires américains** qui ont peur de voir leurs systèmes d'information détruits par l'effet electro-magnétique induit par une explosion nucléaire. Il demande à leurs chercheurs de concevoir un moyen sûr de transporter des informations qui ne dépendrait pas de l'état général physique du réseau, voir même qui supportera la destruction physique partielle tout en continuant d'acheminer les informations.

Officieusement dès les années cinquante au USA, dans le plus grand secret est mis au point un réseau de transmission de données militaires comme le réseau SAGE uniquement réservé aux militaires. Les chercheurs du MIT vont mettre au point en 1969 la commutation de paquets dont nous venons de parler, et concevrons l'architecture distribuée qui sera choisie pour le réseau.

Officiellement, la première installation effective sera connu sous le nom d'**ARPANET** aura lieu en 1970 en raccordant les 4 universités américaines de Santa Barbara, de l'Utah, de Stanford et de Los Angeles. Plusieurs universités américaines s'y raccorderont et continueront les recherches jusqu'en 1974 date à laquelle V.Cerf et R.Kahn proposent les protocoles de base IP et TCP. En 1980 la direction de l'ARPA rendra public les spécifications des ces protocoles IP et TCP. Pendant vingt ans ce réseau a servit aux militaires et aux chercheurs.

Il faut attendre 1990 pour voir s'ouvrir le premier service de fourniture d'accès au réseau par téléphone. Au même moment, ARPANET disparaît pour laisser la place à **Internet**. Un an plus tard, les principes du Web sont établis.

Le world wide web : www

C'est la partie d'Internet la plus connue par le grand public. A l'origine, le World Wide Web (WWW) a été développé en 1990 au CERN, le Centre Européen pour la Recherche Nucléaire, par R.Caillau et T.Berners-Lee. Il autorise l'utilisation de textes, de graphiques, d'animations, de photographies, de sons et de séquences vidéo, avec des liens entre eux fondés sur le modèle hypertextuel.

Le Web est un système hypermédias du genre client/serveur.

C'est sur ces spécifications qu' a été élaboré le langage de description de document du web **HTML** (**H**yper **T**ext **M**arkup **L**anguage).

Pour lire et exécuter ces hypermédias, il faut un logiciel que l'on dénomme un navigateur. Mosaic est l'un des premiers navigateurs Web, distribué gratuitement au public. Depuis 1992, les utilisateurs de micro-ordinateurs peuvent alors se connecter à Internet à partir de leur PC. Internet Explorer de Microsoft et Netscape sont les deux principaux navigateurs les plus utilisés dans le monde.

Les points forts d'Internet :

Il permet à un citoyen de se connecter n'importe où en disposant de :

- Un micro-ordinateur du commerce,
- Un système d'exploitation supportant les protocoles adéquats, tous les SE de micro-ordinateur depuis 1997 disposent d'un moyen simple de se connecter à Internet (Windows, Linux en sont

deux exemples),

- Un modem (se branchant sur une ligne téléphonique ordinaire) à 56000bps ou plus (ADSL) ou bien le câble en attendant de nouveaux produits de transport des signaux.
- Un abonnement chez un fournisseur d'accès à Internet (noeud de communication concentrateur),
- Enfin un navigateur permettant de dialoguer avec les différents serveurs présents sur Internet.

Le revers de médaille d'Internet :

L'inorganisation totale de cette gigantesque et formidable banque de données qu'est un tel réseau mondial qui contient le meilleur et le pire, peut engendrer des dangers pour le citoyen et même pour une démocratie si l'on ne reste pas vigilant.

Enfin, selon les pays, les coûts d'utilisation restent importants (abonnement chez le fournisseur et durée de communication téléphonique pour la connexion), la concurrence des fournisseurs d'accès gratuit permet une baisse du coût général de la connexion. La connexion illimitée et gratuite reste l'objectif à atteindre.

Internet est devenu un problème de société

Trois courants de pensée s'affrontent quant à l'impact d'Internet sur les société humaines :

- **Le courant du tout-Internet** qui prône un nouveau monde virtuel où Internet intervient à tous les niveaux de la vie privée, publique, professionnelle, culturelle voir spirituelle.
- **Le courant des Internetophobes** qui rejette ce nouveau monde virtuel vu comme une accentuation encore plus marquée entre les "riches" et les "pauvres" (la richesse ne s"évaluant plus uniquement en bien matériels, mais aussi en informations).
- **Le courant des "ni-ni"**, ceux qui considèrent que tout outil mérite que l'on s'en serve avec réflexion pour le plus grand nombre, mais qui pensent qu'un outil n'est pas une révolution sociale en lui-même, seul l'homme doit rester au centre des décisions qui le concernent.

La tendance au début du XXI sècle est de renforcer l'aspect commercial (e-business) de ce type de produit sous la poussée des théories ultra-libérales, au détriment de l'intérêt général pour une utilisation plus citoyenne au service de tous.

Intranet

Les entreprises conscientes du danger de pillage, de sabotage et d'espionnage industriel ont repris les avantages de la conception d'Internet en l'adaptant à la notion de réseau local.

C'est le nom d'Intranet qui s'est imposé. Ce genre de réseau local d'entreprise est fondé sur les mêmes techniques, les mêmes procédés qu'Internet, mais fonctionne localement avec un certain nombre d'acteurs bien identifiés :

- Il peut donc être organisé selon la démarche interne de l'entreprise.

- Il n’est accessible qu’aux personnes autorisées si l’entreprise le souhaite.
- Il est connectable à Internet par des passerelles *contrôlées*.
- Il concerne toutes les activités logistiques, commerciales et de communication de l’entreprise.
- Il permet de mettre en œuvre des activités de groupware (travaux répartis par tâches identifiées sur des systèmes informatiques).

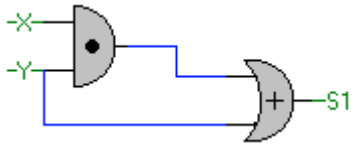
Il peut être organisé en Extranet, permettant la communication entre Intranets de différentes et bien sûr, un Intranet peut être connecté à Internet.

Exercices chapitre1

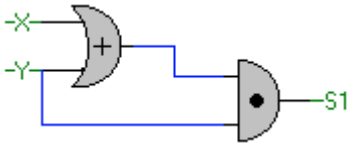
Questions :

On suppose que X, Y et Z sont des variables booléennes, donnez pour chaque circuit ci-dessous l'expression de sa fonction logique S1(X,Y) ou S1(X,Y,Z), évaluez la table de vérité de S1, puis simplifiez par calcul S1.

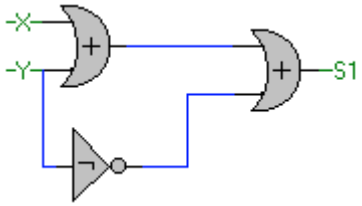
Ex-1 :



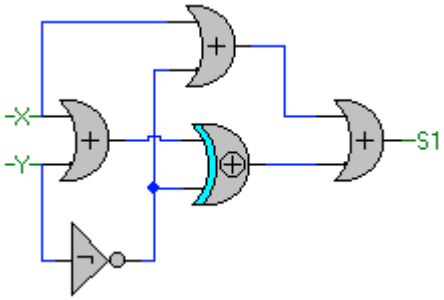
Ex-2 :



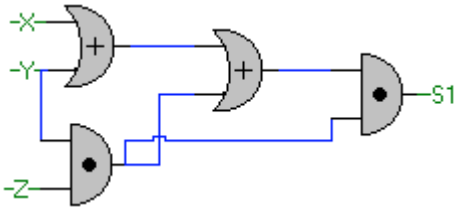
Ex-3 :



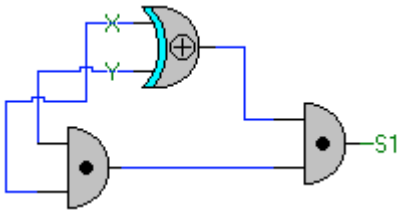
Ex-4 :



Ex-5 :



Ex-6 :



Ex-7 : Lorsque l'on additionne deux entiers positifs en binaire signé dans un ordinateur, le calcul s'effectue dans l'UAL en propageant la retenue sur tous les bits y compris le bit de signe.
Soient deux mémoires à 7 bits Mx et My, contenant respectivement l'entier x et l'entier y représentés en binaire signé avec convention du zéro positif :

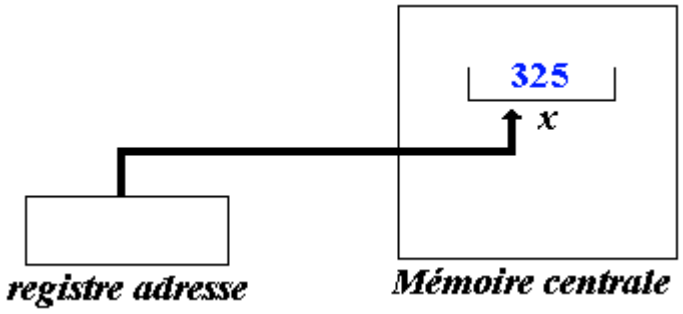
1°) Quel est le nombre entier positif maximal que l'on peut stocker dans Mx ou My ?
2°) Quel est le nombre entier négatif minimal que l'on peut stocker dans Mx ou My ?
3°) Donnez le résultat de l'addition de Mx+My dans une mémoire Mz du même type que Mx et My, dans les deux cas suivants :

3.1) Mx contient l'entier x = 12, My contient l'entier x = 25
3.2) Mx contient l'entier x = 42, My contient l'entier x = 25

Ex-8 : Soit l'entier x = 325 :

1°) Donnez sa représentation en codage binaire pur
2°) Donnez sa représentation en codage Ascii étendu
3°) Quel est en nombre de bits le codage le plus court ?
4°) D'une manière générale pour un entier positif x à n chiffres en représentation décimale donnez un majorant du nombre de bits dans la représentation de x en binaire pur, vérifiez le calcul pour x = 325.

Ex-9 : Soit une mémoire centrale et un registre adresse contenant l'adresse d'une case quelconque de la mémoire centrale :



- On suppose que le registre adresse soit une mémoire à 12 bits.
- On suppose que la taille du mot dans la mémoire centrale est de 16 bits.
- On suppose que la case x contient en binaire en complément à deux le nombre 325.

1°) Combien de mots (de cases) au maximum peut contenir cette mémoire centrale ?
2°) Quel est l'entier positif le plus grand représentable en complément à deux dans un mot de cette mémoire centrale ?
3°) Indiquez si les nombres suivants peuvent être l'adresse d'un mot dans cette mémoire centrale ;

3.1)le nombre 683
3.2)le nombre 2AB
3.3)le nombre 2AB0
3.4)le nombre -5

Réponses :

Ex-1 : S1 = x .y + y simplifiée → S1 = y
Ex-2 : S1= (x + y) .$\bar{y}$ simplifiée → S1 = y
Ex-3 : S1= x + y + $\bar{y}$ simplifiée → S1 = 1
Ex-4 : S1= (x + $\bar{y}$) + (x + y) ⊕ $\bar{y}$ simplifiée → S1 = 1
Ex-5 : S1: (x + y + y.z) . (y.z) simplifiée → S1 = y.z
Ex-6 : S1= (x ⊕ y). (y.x) simplifiée → S1 = 0

Ex-7 : 1°) le nombre maximum = 2^6-1 (soit 63)
2°) le nombre minimum = -2^6(soit -64)
3.1) Mx+My = 37 (car: **0**001100 + **0**011001 = **0**100101)
3.2) Mx+My = -3 (car: **0**101010 + **0**011001 = **1**000011 , le bit de signe a été écrasé)

Ex-8 : 1°) x =101000101
2°) x = 00110011 . 00110010 . 00110101 (car chaque symbole est codé sur 8 bits)
3°) Le codage binaire pur du nombre x occupe 9 bits alors que son codage Ascii occupe 24 bits.
4°) Soit k le nombre de bits (nombre de chiffres binaires) de l'entier x, en décimal x est composé de n chiffres décimaux ⇔ $x < 10^n$, en binaire x se compose de k chiffres binaires ⇔ $x < 2^k$, dès que $2^k \sim 10^n$ ou encore k ~ n $\log_2 10$, soit donc le nombre de bits est de l'ordre de la partie entière du nombre approché n $\log_2 10$, soit k ~ 3,32 . n . Pour un nombre à 3 chiffres k ~ 9, donc 9 bits suffiront pour coder ce nombre en binaire pur.

Ex-9 : 1°) cette mémoire centrale peut contenir au maximum 2^{12} = 4096 mots.
2°) le plus grand entier vaut : 2^{15}-1 (soit 32737).
3°) 683 (oui), 2AB (oui), 2AB (non plus de 12 bits), -5 (non, une adresse est un nombre positif ou nul)

Chapitre 2 : Programmer avec un langage

2.1.Les langages

- Historique des langages de programmation
- Langages procéduraux
- langages fonctionnels
- langages logiques
- langages objets
- langages de spécification
- langages hybrides

2.2.Relations binaires

- Rappel et conventions
- matrice d'une relation binaire
- fermeture transitive d'une relation binaire

2.3.Théorie des langages

- notations et définitions
- grammaire formelle
- classification de Chomsky
- applications - exemples

2.4.Les bases du langage Delphi-Pascal

- structure d'un programme
- les opérateurs
- déclarations des types
- instructions
- fonctions/procédures
- paramètres
- visibilité
- passage par adresse

2.1 : Les langages

Plan du chapitre:

1.Historique des langages

1.1 Les langages procéduraux ou impératifs
1.2 Les langages fonctionnels
1.3 Les langages logiques
1.4 Les langages orientés objets (L.O.O)
1.5 Les langages de spécification
1.6 Les langages hybrides

1. Historique des langages de programmation

La communication entre l'homme et la machine s'effectue à l'aide de plusieurs moyens physiques externes. Les ordres que l'on donne à l'ordinateur pour agir sont fondés sur la notion d'instruction comme nous l'avons déjà vu. Ces instructions constituent un langage de programmation. Depuis leur création, les langages de programmation ont évolué et se sont diversifiés.

Schématiquement il est possible de les classer en cinq catégories :

1° Les langages procéduraux ou impératifs.
2° Les langages fonctionnels.
3° Les langages logiques.
4° Les langages objets.
5° Les langages de spécification.

L'un des principaux objectifs d'un langage de programmation est de permettre la construction de logiciels ayant un minimum de qualités comme la fiabilité, la convivialité, l'efficacité.
Il faut connaître l'histoire des langages et se rendre compte qu'à ce jour, malgré les nouveaux langages du marché et leur efficacité, c'est **Cobol** qui est le plus utilisé (numériquement 200 milliards de lignes Cobol seraient intégrées à des applications existantes [programmez, n°63 Avril 2004] dont 5 milliards de lignes nouvelles chaque année) dans le monde.

L'investissement intellectuel et matériel prédomine sur la nouveauté. Cette remarque est la clef de la compréhension de l'évolution actuelle et future des langages.

Les langages ont fait leurs premiers pas directement sur des instructions machines écrites en binaire, donc rudimentaires sur le plan sémantique. Les améliorations sur cette catégorie de langages se sont limitées à construire des langages symboliques (langage avec mnémonique) et des macro-assembleurs. J.Backus d'IBM avec son équipe a mis au point dès 1956-1958 le premier langage évolué de l'histoire, uniquement conçu pour le calcul scientifique (à l'époque l'ordinateur n'était qu'une calculatrice géante).

Les années 70 ont vu s'éloigner un rêve d'informaticien : parler et communiquer en langage naturel avec l'ordinateur.
Actuellement les langages évolués se diversifient et augmentent en qualité d'abstraction et de convivialité.

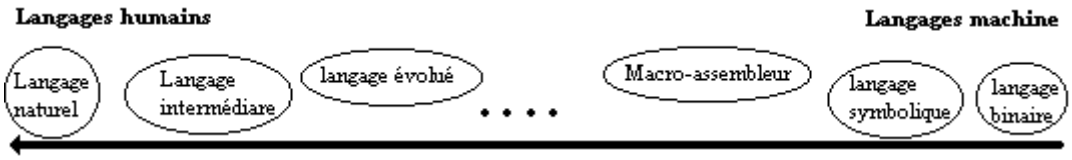


fig : classification sur un axe d'abstraction : de la machine à l'homme

Les langages majoritairement les plus utilisés actuellement sont ceux qui font partie de la catégorie des langages procéduraux ou Hybrides. Les ordinateurs étant des machines de Turing (améliorées par von Neumann), la notion de mémoire machine est représentée par la

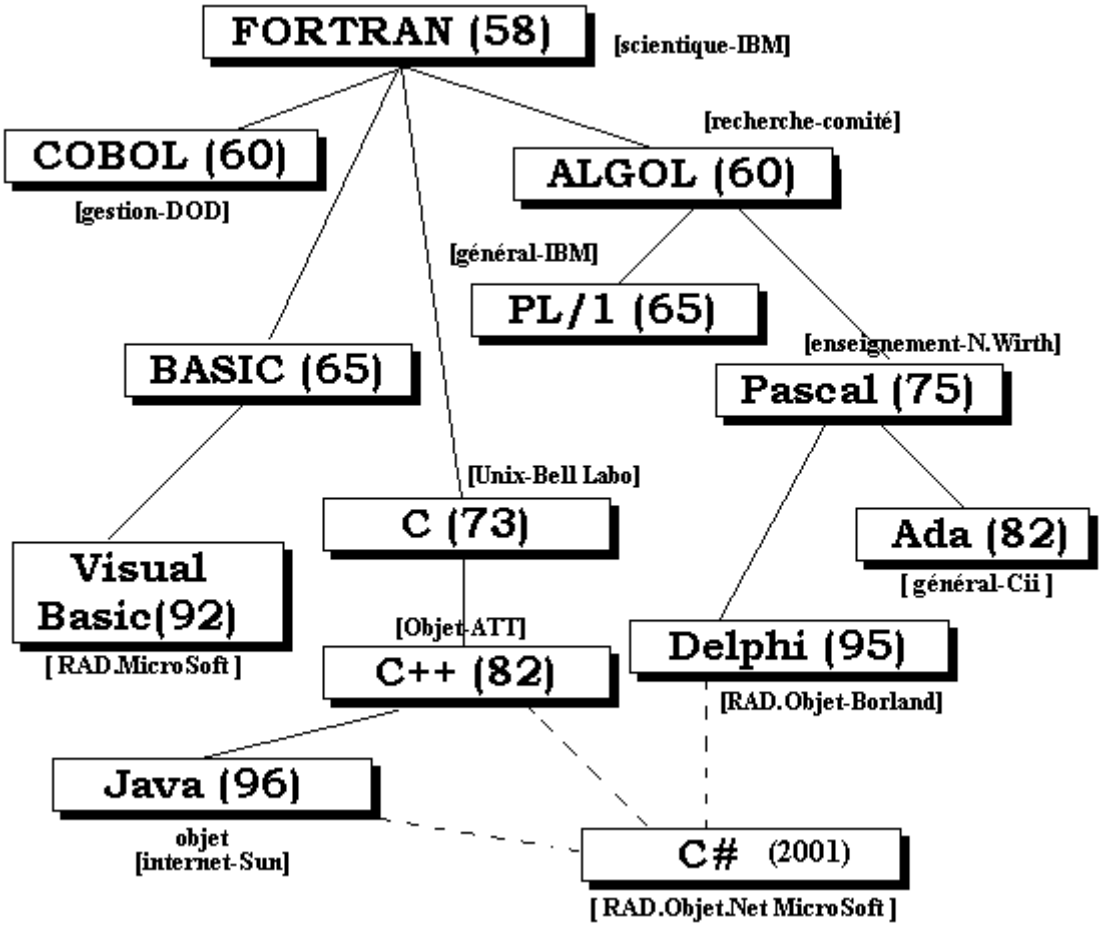
donnée abstraite qu'est une variable, dans un langage procédural. D'autre part, les machines de Türing sont séquentielles et les langages impératifs traitent les instructions séquentiellement. Ceci indique que les langages procéduraux sont parfaitement bien adaptés à l'architecture de l'ordinateur ; ils sont donc plus " facilement " adaptables à la machine.

1.1 Les langages procéduraux ou impératifs

Tous les langages procéduraux ont un ancêtre commun : le langage **FORTRAN**.

Voici un arbre généalogique (non exhaustif) de certains langages connus. Pour chaque langage nous avons indiqué quelques éléments de référence.

Par exemple : **FORTRAN (58) [scientifique - IBM]** signifie que le premier compilateur commercial a été diffusé environ en 1958, que le domaine d'activité pour lequel le langage a été élaboré est le domaine du calcul scientifique, enfin qu'il s'agit d'un acte commercial puisque c'est la compagnie IBM qui l'a fait réaliser.



Dans cette courte liste, seuls **Algol**, **Basic** et **Pascal** sont des langages qui ont été conçus par des équipes dans des buts de recherche ou d'enseignement. Les autres langages sont élaborés par des firmes et des compagnies dans des buts de commercialisation, de rationalisation des coûts de gestion (DOD) etc... Les langages de programmation, comme le reste des outils de la science informatique, sont fortement soumis aux règles du marché, ce qui provoque pour cette discipline le pire et le meilleur.

Pour donner les propriétés des autres catégories de langages, nous nous servirons de la catégorie des langages procéduraux comme référence.

Dans un langage procédural, l'affectation (transfert d'une valeur dans une mémoire) est la base des actions sur les données. La catégorie la plus utilisée après les langages procéduraux est celle des langages fonctionnels.

1.2 Les langages fonctionnels

Dans un langage fonctionnel, les actions reposent sur des fonctions mathématiques ou non qui renvoient des résultats.

Un langage fonctionnel est essentiellement composé d'un dictionnaire de fonctions prédéfinies et d'un mécanisme de construction de nouvelles fonctions par l'utilisateur.

Citons quelques représentants des langages fonctionnels :
LISP :(LISt Processing - 1962) en fait c'est essentiellement un langage de traitement de listes.
SCHEME : c'est un dialecte pédagogique épuré(1975) de LISP.
ML : langage fonctionnel moderne(1990) classé dans la catégorie des langages fonctionnels fortement typés (l'INRIA diffuse gratuitement sur micro-ordinateur une version CAML-Light pour l'enseignement). CAML est utilisé actuellement pour l'enseignement de l'informatique dans les classes préparatoires aux grandes écoles scientifiques françaises.

1.3 Les langages logiques

Citons la catégorie des langages de programmation en logique et son principal représentant :

PROLOG (**PRO**grammation en **LOG**ique - 1982).
Dérivé de l'intelligence artificielle, il oblige le programmeur à penser ses actions en termes de buts et à en faire une description relationnelle (vision déclarative).

Le langage Prolog est fondé sur un moteur d'inférence d'ordre 1 (logique des prédicats), et permet l'exploration exhaustive automatique de différents chemins amenant à des solutions. Il possède une qualité intéressante : il est possible d'interpréter un programme prolog d'une manière *déclarative* ou d'une manière *procédurale*.

Le Groupe d'Intelligence Artificielle de Marseille-Luminy fournit des prologs sur micro-ordinateurs à travers la société PrologIA.

1.4 Les langages orientés objets (L.O.O)

Les langages à objets : ils sont fondés sur une seule catégorie d'éléments : " les objets " qui communiquent entre eux grâce à l'envoi de messages (grâce à des opérateurs appelés méthodes). Par rapport à un langage impératif typé, un objet est l'équivalent (mutatis mutandis) d'une variable (simple ou structurée) et la classe dont il est l'instance correspond au type de la variable.

SIMULA-67 (1967) est le premier langage objet, **SMALLTALK-80**(1980) est un environnement de développement purement objet, **Eiffel**(1990) est un langage objet tourné vers le génie logiciel et la réutilisabilité.

1.5 Les langages de spécification

Les langages de spécification sont encore du domaine de la recherche. Leurs objectifs sont de décrire le plus rigoureusement possible (les modèles principaux sont mathématiques) un logiciel afin de pouvoir le valider et le vérifier.

Nous ne mentionnerons ici que le langage **LPG** de D.Bert(Grenoble) pour les spécifications de types abstraits algébriques, **Z** de J.R. Abrial, le langage dont la notation est fondée sur la théorie des ensembles (puis d'une amélioration de **Z** dénotée **B** par Abrial) et **VDM** langage formel de spécification par pré-condition et post-condition. Ces langages ne peuvent être utilisés d'une manière pratique que sous forme de notation, bien qu'ils soient implantés sur des systèmes informatiques. Ils ne sont pas encore à la disposition du grand public comme les langages des catégories précédentes, bien que certains soient utilisés dans des sites industriels. Par la suite, nous utiliserons un langage de spécification pédagogique fondé sur les types abstraits algébriques.

1.6 Les langages hybrides

Une mention spéciale ici pour des concepts hybrides qui peuvent être de bons compromis entre des catégories différentes. Les concepteurs de tels langages essaient d'importer dans leur langage les qualités inhérentes à au moins deux catégories. La catégorie la plus utilisée est celle des langages impératifs.

Par exemple, la plupart des langages impératifs purs cités plus haut bénéficient d'une " extension " objet, comme **C++** qui est une extension orientée objet du langage **C** conçu à l'origine pour écrire le système d'exploitation Unix.

Plus récemment est apparu un langage comme **Delphi** de Borland qui allie l'approche pédagogique et typée du Pascal, l'approche objet du C++ et les approches visuelles et événementielles de Visual Basic de Microsoft (la sortie fin 2001 de la version entièrement orientée objet de VB, dénommée **VB .Net**, procure à Visual Basic un statut de langage hybride).

Enfin, mentionnons l'important langage **Java** de Sun Microsystems qui permet le développement multi-plateforme en particulier pour l'intranet et qui est grandement utilisé malgré son léger manque de rapidité dû à sa machine virtuelle.

Un mot enfin sur le tout récent langage **C#** support de développement de la plateforme Microsoft .Net, qui a été inventé par le père du langage Delphi (C# s'approprie des avantages de Java et de **Delphi**, il suit de très près la syntaxe de **Java** et celle de **C++**)et qui est le fer de lance de la plateforme .Net de microsoft.

Object Pascal, **C++**, **Ada95**, **Java, C#** sont des langages procéduraux qui ont été fortement étendus ou remaniés pour se conformer aux standards objets.

Remarque de vocabulaire:

L'ordinateur ne " comprenant " que le langage binaire, il lui faut donc un "traducteur" qui lui traduise en binaire exécutable, les instructions que l'humain lui fournit en langage évolué.

Cette traduction est assurée par un programme appelé **compilateur**.

Un compilateur du langage L est donc un programme chargé de traduire un programme "source" écrit en L par un humain, en un programme " cible " écrit en binaire exécutable par l'ordinateur.

2.2 : Relations binaires

Plan du chapitre:

1. Rappel et convention

1. Relation binaire *sur un ensemble*
2. Produit de relations *binaires*
3. Représentation matricielle *d'une relation binaire*
4. *Relation* binaire transposée
5. Matrice du produit *de deux relations*
6. Fermeture transitive *d'une relation binaire*
7. Fermeture réflexo-transitive *d'une relation binaire*
8. Algorithmes *de calcul de matrices*
9. Exemple de calcul *sur une généalogie*

1. Rappels et conventions

Un peu de mathématiques utiles, mais pas trop !

En informatique, la notion de relation est importante. Nous indiquons ici sans rentrer dans les détails que le lecteur trouvera dans des livres spécialisés, en particulier sur la recherche opérationnelle, comment on implante une relation binaire à travers sa matrice de représentation. Ceci peut donc être considéré comme un bon exemple d'application des matrices booléennes en informatique.

Convention

Lorsque nous écrivons " x ← a " ceci se lit: **"x vaut la valeur de a"**.

1. Relation binaire sur un ensemble

Nous appelons relation binaire sur un ensemble E non vide, tout sous-ensemble **R** du produit cartésien E x E.

$$\mathbf{R} \subset E \times E$$

Il est donc possible de définir l'union et l'intersection de deux relations binaires.

2. Produit de relations binaires

Soient ρ et σ deux relations binaires sur un ensemble non vide E. On définit le produit des deux relations $\pi = \rho.\sigma$ ainsi :

$$\forall a, a \in E \quad \forall b, b \in E \qquad a \ \rho.\sigma \ b \ \textbf{ssi} \ \exists c, c \in E / (a \ \rho \ c) \text{ et } (c \ \sigma \ b)$$

Nous énonçons brièvement quelques propriétés de ce produit :

- Le produit est associatif.
- Le produit n'est pas commutatif.

Notations

$\rho^n = \rho.\rho....\rho$ (n fois)
ρ^0 , est la relation telle que : $\forall a, a \in E$ on a toujours $a \ \rho^0 \ a$
$\rho^{n+m} = \rho^n . \rho^m$

3. Représentation matricielle d'une relation binaire

Cas où E est un ensemble fini, c'est d'ailleurs le seul cas qui nous intéresse en informatique où nous ne pouvons pas traiter du non fini.

Soit E l'ensemble : E = { a_1, a_2, ..., a_n }

- Soit ρ une relation binaire sur E.
- Soit M une matrice carrée d'ordre n sur {0,1}. Nous notons (($m_{i,j}$)) l'élément générique de la matrice M.

Nous dirons que M est la matrice de représentation de la relation binaire ρ et nous la noterons Mρ, **ssi** par définition :

```
si ai ρ aj alors mi,j ← 1 sinon mi,j ← 0 fsi
```

Exemple :
E = { 7,8,3 } ; ρ = { (7,8),(7,3),(3,8),(8,7) }
a_1 = 7 ; a_2 = 8 ; a_3 = 3

Voici la matrice Mρ de la relation ρ définie ci-haut :

$$M\rho = \begin{array}{c} \begin{array}{ccc} 7 & 8 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ 7 \\ 8 \\ 3 \end{array}$$

4. Relation binaire transposée

- Soit E l'ensemble : E = {a_1,a_2,...,a_n}
- Soit ρ une relation binaire sur E.

Nous notons ρ^t la relation binaire telle que :

$$\forall a,\ a \in E \quad \forall b,\ b \in E \qquad a\ \rho^t\ b \ \underline{\textbf{ssi}}\ b\ \rho\ a$$

Par construction la matrice de ρ^t est la transposée de la matrice de ρ.

$$M\rho^t = {}^tM\rho$$

5. Matrice du produit de deux relations

En munissant l'ensemble {0,1} d'une structure d'algèbre de boole avec les opérateurs $\wedge$, $\vee$, $\neg$, il nous est possible d'effectuer des calculs sur les matrices de représentation de relations binaires.

- Soient ρ et σ deux relations binaires sur un ensemble non vide E. Soit le produit des deux relations , $\pi = \rho.\sigma$, $M\rho$, $M\sigma$ et $M\pi$ les matrices de ρ, σ et π.
- Soit $((a_{i,j}))$ l'élément générique de $M\rho$.
- Soit $((b_{i,j}))$ l'élément générique de $M\sigma$.

La matrice $M\pi = M\rho.\sigma$ est très exactement par définition le produit booléen en croix des matrices $M\rho$ et $M\sigma$.

$$M\rho \times M\sigma = M\pi = M\rho.\sigma = \left[\bigvee_{k=1}^{n} (a_{i,k} \wedge b_{k,j})\right]$$

6. Fermeture transitive d'une relation binaire

- Soit E l'ensemble : $E = \{ a_1, a_2, \ldots ,a_n \}$
- Soit ρ une relation binaire sur E.

Nous posons par définition sa fermeture transitive qui est la relation binaire ρ^+ :

$\rho^+ = \bigcup_{n=1}^{\infty} \rho^n$, en fait dans le cas où E est fini l'union se limite à un nombre fini **k** de ρ^n distincts donc :

$$\rho^+ = \bigcup_{n=1}^{k} \rho^n$$

En informatique, les ensembles sont toujours finis donc nous considérons que la fermeture transitive de ρ s'écrit :

$$\rho^+ = \rho^1 \cup \rho^2 \cup \ldots \cup \rho^{k-1} \cup \rho^k$$

7. Fermeture réflexo-transitive d'une relationinaire

- Soit E l'ensemble : $E = \{a_1, a_2, \ldots, a_n\}$
- Soit ρ une relation binaire sur E, sa fermeture transitive $\rho+$.

On note par définition $\rho*$ sa fermeture réflexo-transitive :

$$\rho^* = \rho^+ \cup \rho^0$$

Remarque

Soit un couple (a,b) de E x E tel que a ρ^* b :
a ρ^* b $\Leftrightarrow \exists$ n , n $\in$ E / a ρ^n b

Nous dirons dans ce cas qu'il existe " un chemin de longueur n, allant de a vers b ". En effet d'après la définition du produit :

a ρ^* b $\Leftrightarrow \exists (c_1, \ldots, c_n), \forall k, k \in [1, n], c_k \in E$
tels que : $(a \rho c_1)$ et $(c_1 \rho c_2)$ ...et $(c_n \rho b)$

8. Algorithmes de calcul de matrices

Calcul de la matrice produit à partir de la formule :

$$M\rho \times M\sigma = M\pi = M\rho.\sigma = [\bigvee_{k=1}^{n} (a_{i,k} \wedge b_{k,j})]$$

Notons $((m_{i,j}))$ l'élément générique de la matrice produit, voici le corps d'un algorithme de calcul de la matrice produit :

```
pour i ← 1 jusquà n faire
 pour j ← 1 jusquà n faire
  S ← 0 ;
  pour k ← 1 jusquà n faire
   S ← S ∨ (a_{i,k} ∧ b_{k,j})
  Fpour ;
  m_{i,j} ← S
 Fpour
Fpour
```

Algorithme de Warshall pour le calcul de la fermeture transitive :

Avec les mêmes notations de l'algorithme précédent,soit $((a_{i,j}))$ l'élément générique de la matrice Mρ, l'algorithme de Warshall calcule $M\rho^+$:

```
pour k ← 1 jusquà n faire
 pour i ← 1 jusquà n faire
  pour j ← 1 jusquà n faire
   a_{i,j} ←a_{i,j}∨ (a_{i,k} ∧ a_{k,j})
  Fpour ;
 Fpour
Fpour
```

9. Exemple de calcul sur une généalogie

E = l'ensemble des individus d'une même famille depuis plusieurs générations.

Soient **r**, **s** et **t** les relations binaires :

- x **r** y **ssi** x **est le père de** y
- x **s** y **ssi** x **est la mère de** y
- x **t** y **ssi** x **est un enfant de** y

On peut définir les liens familiaux à l'aide des opérations sur les relations binaires :

r^2 = **est grand père paternel de**
s^2 = **est grand mère maternelle de**
r.s = **est grand père maternel de**
s.r = **est grand mère paternelle de** (*non commutativité évidente !)*
$r \cup s$ = **est parent de**
r^n = **est arrière arrière...arrière grand père paternel de**
$(r \cup s)^+$ = **est un ancêtre de** (on voit ici la signification pratique de la fermeture transitive qui relie deux individus par un chemin d'ascendants dans son arbre généalogique)
$u.u^t$ = **est frère ou sœur de** etc ....

2.3 : Théorie des langages

Plan du chapitre:

1. Notations et définitions générales

2. Grammaire formelle ou algébrique

2.1 Monoïde
2.2 Grammaire formelle
2.3 Opérations sur les mots
2.4 Langage engendré par une grammaire
2.5 Grammaire d'états finis
2.6 Arbre de dérivation d'un mot
2.7 Diagrammes syntaxiques

3. Classification de Chomsky des grammaires

3.1 Les grammaires syntaxiques
3.2 Les grammaires sensibles au contexte
3.3 Les grammaires indépendantes du contexte
3.4 Les grammaires d'états finis ou de Kleene

4. Applications et exemples

4.1 Expressions arithmétiques : une grammaire ambiguë
4.2 Expressions arithmétiques : une grammaire non ambiguë

1. Notations et définitions générales

Un langage est fait pour communiquer. Les humains doivent communiquer avec les ordinateurs : ils ont donc élaboré les bases d'une théorie des langages. Dans ce chapitre nous donnons les fondements formalisés d'une telle théorie autour de la notion de grammaire formelle.

Remarque et convention :

- Certains éléments d'un langage s'appellent les symboles.
- Soit S un ensemble de symboles($S \neq \emptyset$). Ce sont les éléments indécomposables dans ce langage (c'est-à-dire non exprimables en autres symboles du langage).

Définition *expression sur S*

On appelle expression sur S, toute suite finie de symboles de S.

e : [1, **n**]→ S , **e** est une expression sur S, **n** est un entier naturel, **n** ≥ 1.

(*e est alors un métasymbole décrivant l'expression S*).

Notation:

On désigne **e** par : **e** = $s_1 s_2 s_3 \ldots s_n$, **n** ≥ 1 où : k, 1 ≤ k ≤ n , $s_k \in S$
et par définition e (k) = s_k (k ∈[1, n]).

On note S+ = { e / ∀e, e expression sur S }
S+ est l'ensemble de toutes les expressions formées sur S.

Définissons deux opérations sur S+ :

L'égalité d'expressions

Soient **e1** et **e2** deux expressions sur S, on définit leur égalité ainsi :

e1 = e2 **ssi** ∃k , k ≥ 1
e1 : [1, k]→S
e2 : [1, k]→S
∀i, 1 ≤ i ≤ k **e1**(i) = **e2**(i)

la concaténation d'expressions

soient **e**$\in$S+ et **f** $\in$ S+, on construit le "produit" des deux expressions e.f :

e : [1, n] $\rightarrow$ S
f : [1, p] $\rightarrow$ S
e.f: [1, n+p] $\rightarrow$ S

avec :
e.f(i) = e(i) **ssi** i $\in$ [1, n]
e.f(i) = f(i) **ssi** i $\in$ [n+1, n+p]

Notation : (la concaténation de 2 expressions sur S)

Soient **e** et **f** deux expressions :
e = $s_1s_2s_3...s_n$
f = $t_1t_2t_3...t_p$

e.f est notée : $s_1s_2s_3...s_n\ t_1t_2t_3...t_p$

2. Grammaire formelle ou algébrique

Comme dans les langages naturels, les informaticiens ont, grâce aux travaux de N.Chomsky, formalisé la notion de grammaire d'un langage informatique.

2.1 Monoïde

A) Soit A un ensemble fini appelé alphabet ainsi défini :

$A = \{ a_1 ,..., a_n \}$ ($A \neq \varnothing$)

Notations :

$A^1 = A$
$A^2 = \{ x_1x_2 / (x_1 \in A) \text{ et } (x_2 \in A) \}$
$A^3 = \{ x_1x_2x_3 / (x_1 \in A) \text{ et } (x_2 \in A) \text{ et } (x_3 \in A) \}$
.....
$A^n = \{ x_1x_2....x_n / \ \forall i, 1 \leq i \leq n , (x_i \in A) \}$

convention

$A^0 = \{ \varepsilon \}$ (appelé séquence vide)

B) On note A^* et A^+ les ensembles suivants :

$A^* = \bigcup_{n=0}^{\infty} A^n$ $A^+ = A^* - \{ \varepsilon \} = A^* - A^0$	$A^+ = \bigcup_{n=1}^{\infty} A^n$

On définit sur A^* une loi de composition interne appelée concaténation, notée $\bullet$:

$(x, y) \rightarrow x \bullet y = xy$ *(noms des symboles accolés)*

La concaténation possède les propriétés suivantes :

- La loi $\bullet$ est associative :
  $(x \bullet y) \bullet z = x \bullet (y \bullet z)$
- l'élément ε est un élément neutre pour la loi $\bullet$:
  $\forall x \in A^*, x \bullet \varepsilon = \varepsilon \bullet x = x$

Définition :

$(A^*, \bullet)$ est un monoïde libre

2.2 Grammaire formelle

Notations :

Un alphabet est aussi appelé **vocabulaire** ; une **chaîne** ou un **mot** est un élément d'un monoïde ; la **longueur** d'un mot **x** (ou chaîne) est le nombre d'éléments du vocabulaire qui le compose et elle est notée habituellement |**x**|.

Exemple : Vocabulaire V = { a , b }
x = aaabbaab , $x \in V^*$ et |**x**| = 8

Remarque :

On note $|\mathbf{x}|_a$ le nombre de symboles " a " du vocabulaire V composant le mot **x**.
x = aaabbaab $\Rightarrow |\mathbf{x}|_a = 5$ et $|\mathbf{x}|_b = 3$

Définition : C-Grammaire

On appelle **C-Grammaire** (ou, grammaire algébrique de type 2) tout quadruplet :

$G = (V_N, V_T, S, R)$ où :
V_N est un vocabulaire non terminal ou auxiliaire ($V_N \neq \varnothing$)
V_T est un vocabulaire terminal ($V_T \neq \varnothing$)
$S \in V_N$, un élément particulier appelé axiome de G
$V_N \cap V_T = \varnothing$
$R \subset (V_N \cup V_T)^* \times (V_N \cup V_T)^*$, R *est un sous-ensemble fini*

Notations :

- R est appelé l'ensemble des règles de la grammaire G ;
- Une règle $r_i \in R$ est de la forme (A , α) / [$A \in V_N$ et $\alpha \in (V_N \cup V_T)^*$] ,
  Elle est notée : $\mathbf{r_i : A \rightarrow \alpha}$
- Lorsque $\alpha \in V_T^*$, la règle $r_i : A \rightarrow \alpha$, est dite **règle terminale**.

Nous ne considérerons par la suite que les grammaires dites de type 2 encore appelées grammaires indépendante du contexte (Context Free), dans lesquelles les règles ont la forme suivante :

$$\mathbf{R \subset V_N \times (V_N \cup V_T)^*}$$

2.3 Opérations sur les mots

Soit G une C-Grammaire, $G = (V_N, V_T, S, R)$. On définit sur $(V_N \cup V_T)^*$ une relation binaire appelée " **dérivation directe** " notée $\Rightarrow$ définie comme suit :

Définition : dérivation directe

Soient $a \in (V_N \cup V_T)^*$ et $b \in (V_N \cup V_T)^*$

On note $a \Rightarrow b$ et l'on dit que **b** dérive directement de **a**, ou que **a** se dérive directement en **b**, si et seulement si

1°) $\exists\ \alpha \in (V_N \cup V_T)^*$
2°) $\exists\ \beta \in (V_N \cup V_T)^*$
3°) $\exists\ r_i \in R$, telle que : $r_i : A_i \rightarrow \gamma$

4°) a et b s'écrivent :
$a = \alpha\ A_i\ \beta$
$b = \alpha\ \gamma\ \beta$

Notation :

> On emploie aussi les termes de " **règle de réécriture** "
> ou de " **règle de dérivation** ".

Nous obtenons un processus de construction des mots de la grammaire G par application de la dérivation directe. Si l'on réitère plusieurs fois ce processus de dérivation directe, on obtient à partir d'un mot, une suite de mots de G. En fait il s'agit de construire la fermeture transitive de la relation binaire $\Rightarrow$. Cette nouvelle relation est appelée la **dérivation dans G** (la dérivation directe en devenant un cas particulier)

Définition : dérivation

On dit que **x** se dérive en **y**, s'il existe une suite finie de dérivations directes permettant de passer de **x** à **y** :
(**x**, x_0, x_1,...,x_n et **y**) étant des mots de $(V_N \cup V_T)^*$on a le chemin suivant :

$$\mathbf{x} \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \ldots x_n \Rightarrow \mathbf{y}$$

on écrit : $x \Rightarrow^* y$, que l'on lit : **x** se dérive en **y.**

$\Rightarrow^*$ est la fermeture transitive de la relation binaire $\Rightarrow$

2.4 Langage engendré par une grammaire

Nous nous intéressons maintenant à toutes les dérivations possibles construites dans G, par application des règles de G, en privilégiant un point de départ unique pour chacune des dérivations.

Nous avons vu que chaque règle de G commençait en partie gauche par un élément de V_N. Nous construisons alors toutes les productions ayant comme point de départ S l'axiome de G. L'ensemble **L** de tous les mots construits s'appelle le " **langage engendré par la grammaire G** " **:** $L \subset V_T^*$.

Définition : langage engendré

Soit la C-grammaire G, $G = (V_N , V_T , S , R)$

L'ensemble $L(G) = \{ \mathbf{u} \in V_T^* / S \Rightarrow^* \mathbf{u} \}$
s'appelle le **langage engendré** par G.

Exemple grammaire G_0 :

G_0 : $V_N = \{ S \}$,$V_T = \{a,b\}$
Axiome : S
Règles
1 : $S \rightarrow aSb$
2 : $S \rightarrow ab$

Le langage engendré par G_0 est :

$$\mathbf{L(G_0) = \{ a^n b^n \ / \ n \geq 1 \}}$$

2.5 Grammaire d'états finis

Ce sont des C-Grammaires dans lesquelles les parties droites de règles ont une forme particulièrement simple (on classifie d'ailleurs les grammaires algébriques en général en 4 types en fonction de la forme de leurs règles.

Les C-grammaires sont dites de type-2 et les K-grammaires ou grammaires de Kleene sont dites de type-3).

Pour une grammaire de type-3 ou K-grammaire les règles sont de 2 formes :

A $\rightarrow$ **a** **(a** $\in V_T$**)**
ou bien
A $\rightarrow$ **aB** **(B** $\in V_N$ et **B** pouvant être égal à **A)**

Exemple :

G_1 : $V_N = \{ S,A\}$
$V_T = \{a,b\}$
Axiome : S
Règles
1 : $S \rightarrow a\,S$
2 : $S \rightarrow aA$
3 : $A \rightarrow bA$
4 : $A \rightarrow b$

Le langage engendré par G_1 est :

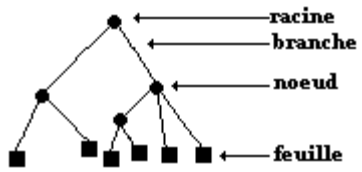
$$\mathbf{L(G_1) = \{ a^n b^p \ / \ (n \geq 1) \text{ et } (p \geq 1) \}}$$

2.6 Arbre de dérivation d'un mot

On appelle **arbre** ***A*** toute structure sur un ensemble E qui est :

- soit une structure vide notée ***A***,
- soit une élément noeud **r** associé à un nombre fini d'arbres disjoints vides ou non : $A_1, A_2, \ldots, A_n$.
- notation : $A = < r, A_1, A_2, \ldots, A_n >$

Représentation graphique d'un arbre :



Un arbre est dit " **étiqueté** " si l'on nomme (attribution d'un symbole de nom) sa racine et ses noeuds.

Définition : arbre de dérivation

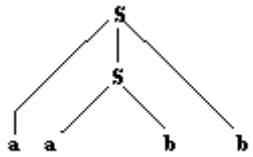
Soit la C-grammaire G, G = (V_N, V_T, S, R).

Un arbre étiqueté est un " **arbre de dérivation** " dans G **ssi** :

- L'alphabet des étiquettes est inclus dans $V_N \cup V_T$.
- Les noeuds sont étiquetés par des éléments de V_N.
- Les feuilles sont étiquetées par des éléments de V_T.
- L'étiquette de tout noeud est un élément de V_N.

Pour tout noeud < **A**, f_1, f_2, ..., f_n> on associe une règle R de la forme : **A** $\rightarrow f_1 f_2 \ldots f_n$ (règle de dérivation dans G).

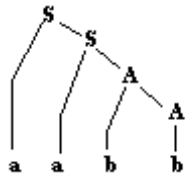
Exemples :



mot a^2b^2 dans G_0

Règles de G_0 appliquées :

$$S \rightarrow^1 aSb \rightarrow^2 aabb$$



mot a^2b^2 dans G_1

Règles de G_1 appliquées :

$$S \rightarrow^1 aS \rightarrow^2 aaA \rightarrow^3 aabA \rightarrow^4 aabb$$

Définition : grammaire ambiguë

Une grammaire est dite **ambiguë** si une chaîne a au moins deux arbres de dérivation différents dans G.

Exemple de grammaire ambiguë :

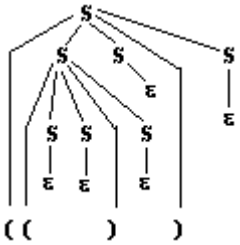
G2 : V_N = {S}
V_T = {(,)}
Axiome : S
Règles
1 : S → (SS)S
2 : S → ε

Le langage engendré par G_2 $L(G_2)$ se dénomme langage des parenthèses bien formées.

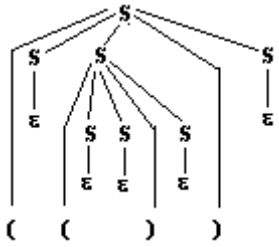
$L(G_2)$ = { ( ) , (( )(( ))( )), ( )( ), ….. }

Soit le mot (( )) de G_2 , voici 2 arbres de dérivation de (( )) dans G_2 :

Arbre 1 :



Arbre 2 :

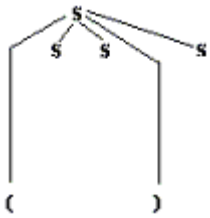


Arbre 1 correspond dans G_2 à la suite des dérivations suivantes :

(afin d'y voir plus clair entre les S choisis nous soulignons les symboles S dérivés)

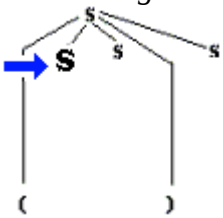
on part de l'axiome **S** *et l'on applique la règle 1:*

$S \rightarrow^{1} (SS) S$



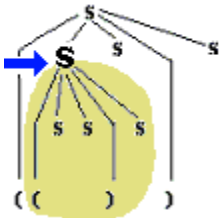
on applique la règle 1 au premier ***S*** *de gauche :*

$S \rightarrow^{1} (SS) S$



Ce qui donne :

$S \rightarrow^{1} (\underline{S}S) S \rightarrow^{1} ((S S) S S) S$



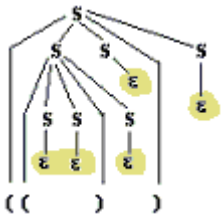
puis on dérive tous les symboles S à partir de la règle 2 :

Règle 2

$S \rightarrow \varepsilon$

appliquée 5 fois à :

((S S) S S) S

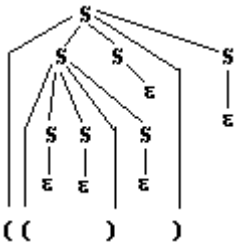


$S \rightarrow \ldots \rightarrow^{2} ((\varepsilon \varepsilon \ )\varepsilon \varepsilon \ ) \ \varepsilon$

Le symbole ε est un élément neutre ($x\varepsilon = \varepsilon x = x$) , nous avons donc comme production finale de cette suite de dérivation le mot : $((\varepsilon \varepsilon) \varepsilon \varepsilon) \varepsilon = (( ))$.

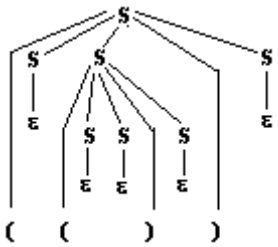
En conclusion, le mot **(( ))** dérive de l'axiome S :

$$S \Rightarrow^{*} (( ))$$



Arbre1 : (( )) est un arbre de dérivation de mot dans la grammaire G_2 .

Arbre 2 correspond dans G_2 à la suite des dérivations suivante :

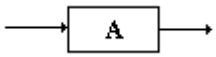
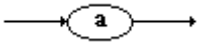
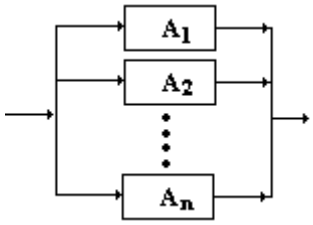
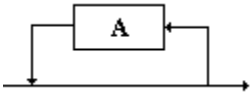
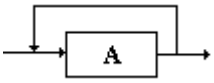


$S \rightarrow^{1} (S \ S) \rightarrow^{1} (S (S S) S) S \rightarrow^{2} \ldots \rightarrow^{2} (\varepsilon (\varepsilon \ \varepsilon) \ \varepsilon) \ \varepsilon = (( ))$.

Le mot **(( ))** dérive de l'axiome S une seconde fois avec un autre arbre de dérivation distinct du précédent, donc la grammaire G_2 est effectivement ambiguë.

2.7 Diagrammes syntaxiques

Il est possible de représenter graphiquement les règles de dérivation d'une grammaire formelle par des diagrammes dénotés " diagrammes syntaxiques ". Cette représentation graphique a pour effet de condenser l'écriture des règles et d'autoriser une meilleure lisibilité.

REGLES	DIAGRAMMES
A $\in V_N$	
a $\in V_T$	
B $\rightarrow \varepsilon$	
B $\rightarrow$ **A1** **B** $\rightarrow$ **A2** ou encore : **.....** **B** $\rightarrow$ **A1|...|An** **B** $\rightarrow$ **An**	
B $\rightarrow$ **AB** | ε ou : **B** $\rightarrow$ **{A}***	
B $\rightarrow$ **AB** | **A** ou: **B** $\rightarrow$ **{A}**$^+$	

3. Classification de Chomsky des grammaires

Traditionnellement les grammaires algébriques sont classables en quatre catégories qui se différentient par la forme de leurs règles.

Elles sont notées par leur type (type 0, type 1, type 2, type 3). Il existe une relation d'inclusion provenant de leurs définitions :

type 3 $\subset$ **type 2** $\subset$ **type 1** $\subset$ **type 0**

3.1 Les grammaires syntaxiques - type 0

Les règles ont la forme générale suivante : $\alpha \rightarrow \beta$

pour une règle $\alpha \rightarrow \beta$, les symboles (α, β) doivent être de la forme :

$$\alpha \in (V_N \cup V_T)^+$$
$$\beta \in (V_N \cup V_T)^*$$

3.2 Les grammaires sensibles au contexte - type 1

Les règles ont la forme suivante : $\alpha A \beta \rightarrow \alpha \gamma \beta$

pour une règle $\alpha A \beta \rightarrow \alpha \gamma \beta$, les symboles($\alpha$, β, γ **,** A**)** doivent être de la forme :

$$A \in V_N$$
$$\alpha \in (V_N \cup V_T)^*$$
$$\beta \in (V_N \cup V_T)^*$$
$$\gamma \in (V_N \cup V_T)^+$$

3.3 Les grammaires indépendantes du contexte - type 2

Les règles ont la forme suivante : $A \rightarrow \alpha$

Pour une règle $A \rightarrow \alpha$, les symboles(α,A) doivent être de la forme :

$$\alpha \in (V_N \cup V_T)^*$$
$$A \in V_N$$

3.4 Les grammaires d'états finis ou de Kleene - type 3

Les règles n'ont que deux formes possibles :

$A \rightarrow a$ *ou bien* $A \rightarrow aB$

Pour ces règles, les symboles (a, A, B) doivent être de la forme :

A $\in V_N$
B $\in V_N$
a $\in V_T$

4. Applications et exemples

4.1 Expressions arithmétiques : une grammaire ambiguë

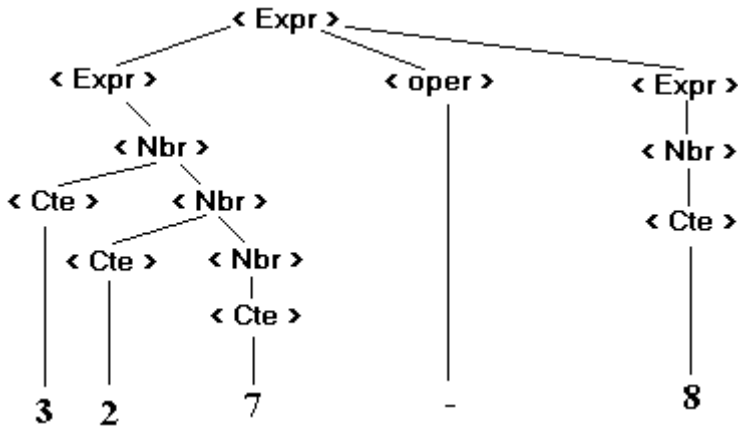
Soit la grammaire G_{exp} = (V_N,V_T,Axiome,Règles)

V_T = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(** }
V_N = {< Expr >, < Nbr > , < Cte > , < Oper > }
Axiome : < Expr >

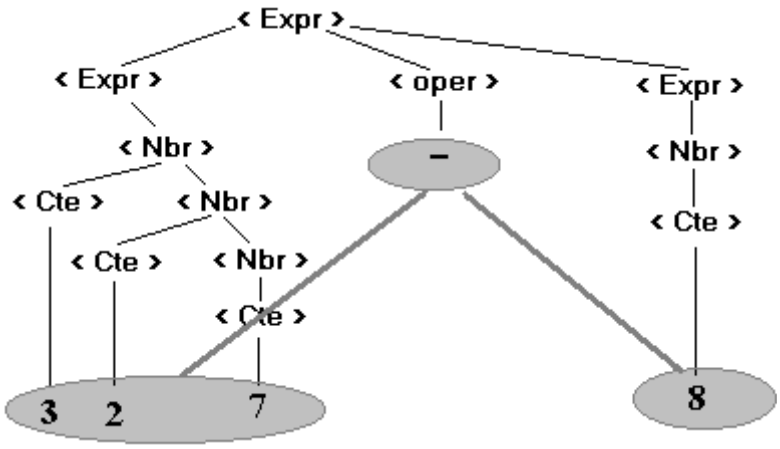
Règles :
1 : < Expr > → < Nbr > | **(**< Expr >**)** | < Expr >< Oper >< Expr >
2 : < Nbr > → < Cte > | < Cte >< Nbr >
3 : < Cte > → **0** | **1** |...| **9**
4 : < Oper > → **+** | **-** | ***** | **/**

Les mots de L(G_{exp}) sont des expressions de la forme (x+y-z)*x etc... où x, y, z sont des entiers.

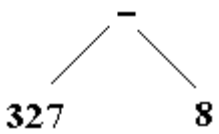
Exemple : **327 - 8** est un mot de L(G_{exp})
Ce mot n'a qu'un seul arbre de dérivation dans G_{exp}, dessinons son arbre de dérivation :



Nous pouvons faire ressortir les liens abstraits qui relient les trois éléments terminaux 327, - et 8 :



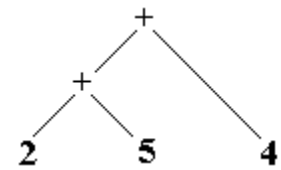
L'arbre obtenu en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 ".



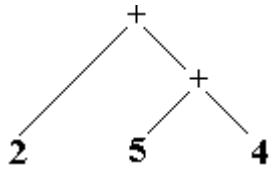
Cet arbre abstrait permet de manipuler la structure générale du mot facilement tout en résumant la structure générale de l'arbre de dérivation.

Soient trois autres mots de $L(G_{exp})$ **2+5+4**, **2-5+4** et **2+5*4**, ils ont chacun deux arbres de dérivation. Nous donnons ci-après deux arbres abstraits de chaque mot.

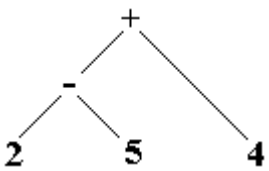
Arbre-1 : 2+5+4



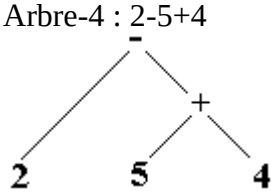
Arbre-2 : 2+5+4



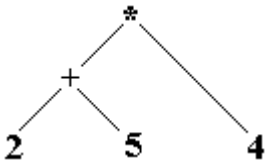
Arbre-3 : 2-5+4



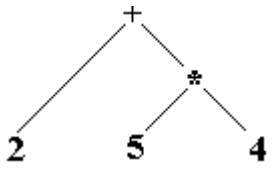
Arbre-4 : 2-5+4



Arbre-5 : 2+5*4



Arbre-6 : 2+5*4



Nous remarquons donc que G_{exp} est une grammaire ambiguë puisqu'il existe un mot possédant au moins deux arbres de dérivation.

Pour l'instant les mots **2+5+4**, **2-5+4** et **2+5*4** ne sont que des concaténations de symboles sans aucun sens particulier

Si nous voulions aller plus loin en donnant un sens (de la **sémantique**) à ces mots de telle façon qu'ils représentent des calculs sur les entiers avec les propriétés classiques des opérations sur les entiers, nous pourrions nous trouver un " *bon choix* " parmi les arbres abstraits précédents.

Nous appellerons ces choix " interpréter " l'expression.

Examen de la situation pour le mot 2+5+4 :

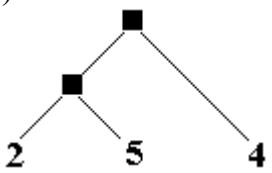
- Arbre-1 s'interprète : (2+5)+4
- Arbre-2 s'interprète : 2+(5+4)

L'opérateur " + " est associatif donc pour notre interprétation les deux arbres 1 et 2 peuvent convenir.

Examen de la situation pour le mot 2-5+4 :

- Arbre-3 s'interprète : (2-5)+4
- Arbre-4 s'interprète : 2-(5+4)

Les opérateurs + et - sont de même priorité et nous obtenons deux expressions différentes selon le choix de l'arbre.
Traditionnellement lorsque deux opérateurs ont la même priorité, l'évaluation se fait à partir de la gauche de l'expression. Donc l'arbre 3 conviendrait.
Nous pourrions penser lever l'ambiguïté en choisissant systématiquement l'arbre abstrait d'évaluation à gauche correspondant à un parenthésage implicite à gauche(comme arbre-1 et arbre-3) :



Nous allons voir ci-dessous que ce n'est pas possible.

Examen de la situation pour le mot 2+5*4 :

- Arbre-5 s'interprète :(2+5)*4
- Arbre-6 s'interprète : 2+(5*4)

Les opérateurs + et * n'ont pas la même priorité. Nous obtenons deux expressions différentes selon le choix de l'arbre. Mais ici c'est le choix de l'arbre 6 qui s'impose à cause de la priorité du * sur le +.

Nous avons fait ressortir le fait qu'il était impossible de privilégier systématiquement pour " l'interprétation " des expressions une catégorie d'arbre plutôt qu'une autre, il faut donc changer de grammaire et éviter l'ambiguïté.

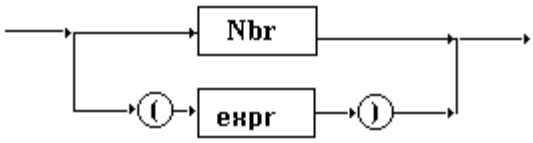
4.2 Expressions arithmétiques : une grammaire non ambiguë

Nous donnons ci-dessous une grammaire non ambiguë basée sur la précédente et tenant compte de la précédence (priorité d'opérateur). Nous séparons les opérateurs en deux catégories ; les opérateurs de priorité zéro (Oper_0) et ceux de priorité un (Oper_1).

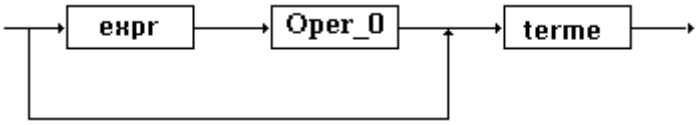
V_T = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(** }
V_N = {< Expr >, < Nbr > , < Cte > , < Oper_0 > , < Oper_1 >, < facteur >, < terme > }
Axiome : < Expr >

Règles :

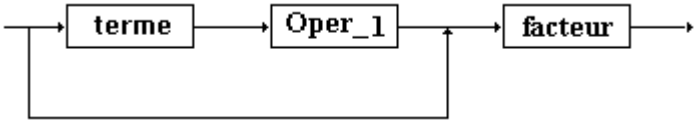
facteur



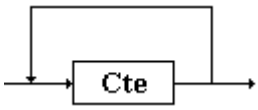
expr



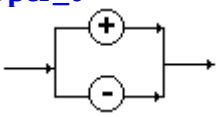
terme



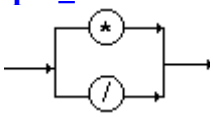
Nbr



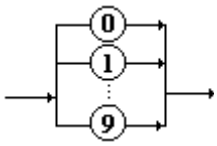
Oper_0



Oper_1



Cte



En pratique ce ne sera pas une telle de grammaire qui sera retenue pour le calcul des expressions arithmétiques car elle contient une règle récursive gauche, ce qui la rend difficilement analysable par des procédés simples.

2.4 : Une grammaire du Pascal

Plan du chapitre:

1. Rappel de la structure d'un programme Pascal

2. Les opérateurs en pascal

2.1 Les opérateurs multiplicatifs
2.2 Les opérateurs additifs
2.3 Les opérateurs relationnels
2.4 Déclarations des constantes

3. Déclarations des types en Pascal

3.1 Déclarations des types simples
3.2 Déclarations des types structurés

4. Instructions en Pascal

4.1 Instruction d'affectation
4.2 Instruction de condition
4.3 Instruction d'itération while…do
4.4 Instruction d'itération repeat…until
4.5 Instruction d'itération for…to
4.6 Instruction case…of

5. Fonctions et procédures en Pascal

6. Paramètres en Pascal

6.1 Lecture seulement : passage par valeur
6.2 Accès direct : passage par adresse ou par référence

7. Fonction ou procédure ?

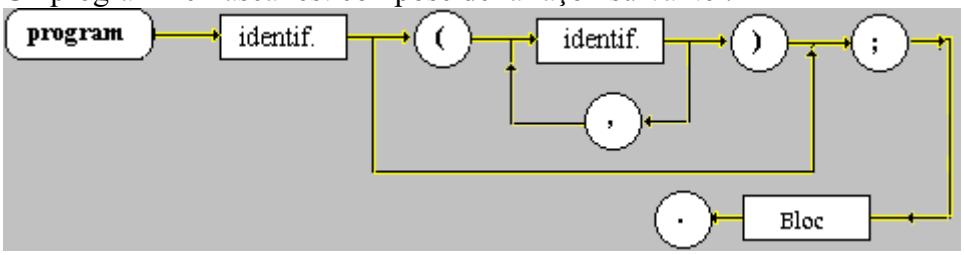
8. Visibilité des variables

1. Rappel de la structure d'un programme Pascal

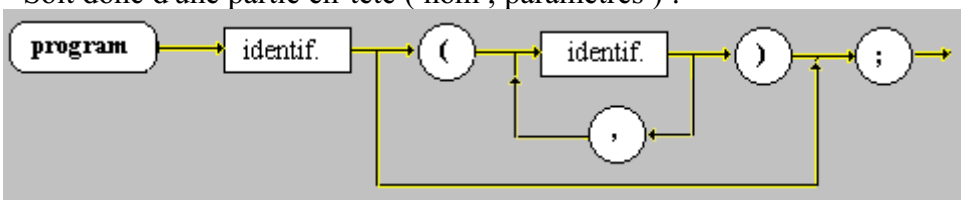
*Il ne s'agit pas d'apprendre le langage pascal, mais plutôt d'un résumé visant à se remémorer les principes de base du langage, et ainsi de se familiariser avec les principes utiles et pratiques du langage relativement à la programmation. La société Borland-Inprise met sur son site web, gratuitement par téléchargement, des compilateurs pascal anciens mais efficaces pour le débutant (**http://www.borland.fr**).*

Nous utilisons une description d'un Pascal-Delphi réduit à l'aide des diagrammes syntaxiques.

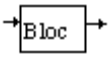
Un programme Pascal est composé de la façon suivante :



- Soit donc d'une partie en-tête (nom , paramètres) :



- d'une partie corps (ou Bloc) :



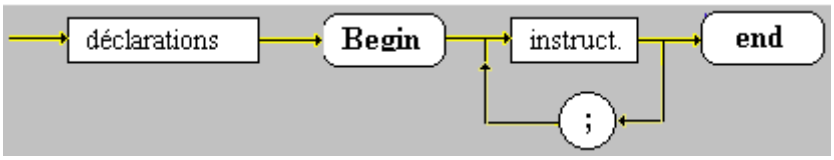
- et se termine par un point :

Exemples d'en-tête :

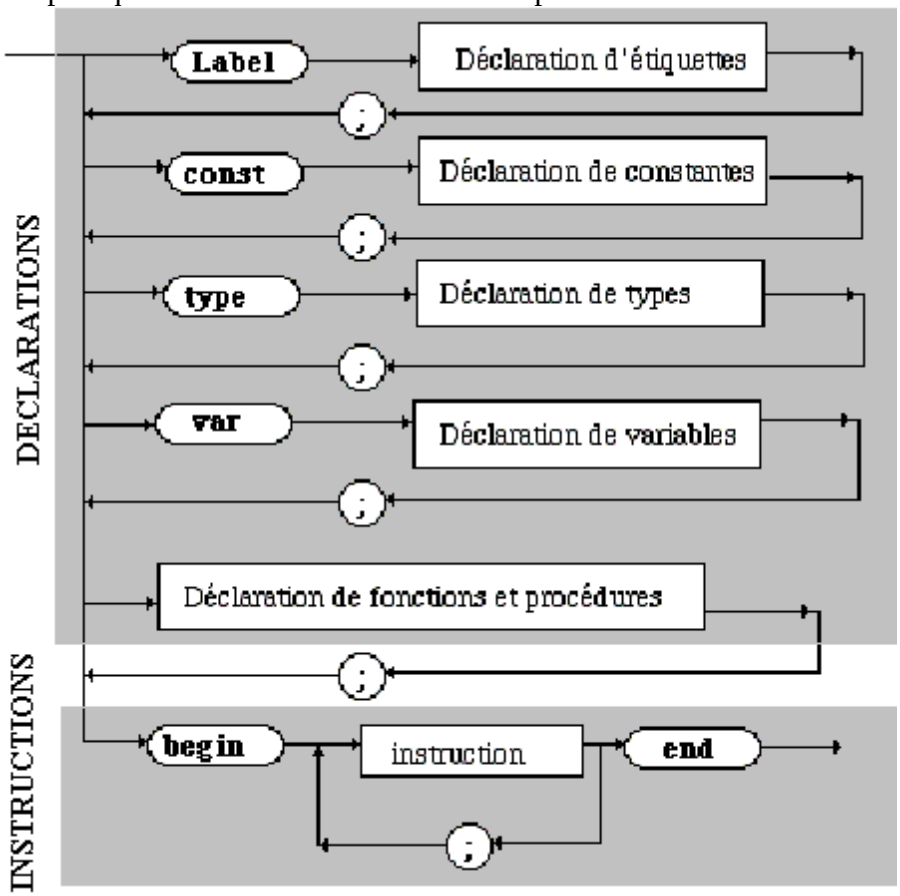
1°) **program** exemple_01 (input, output) ;
2°) **program** exemple_02 ;

Le langage Pascal étant structuré, un bloc est composé de sections ou paragraphes bien séparées :

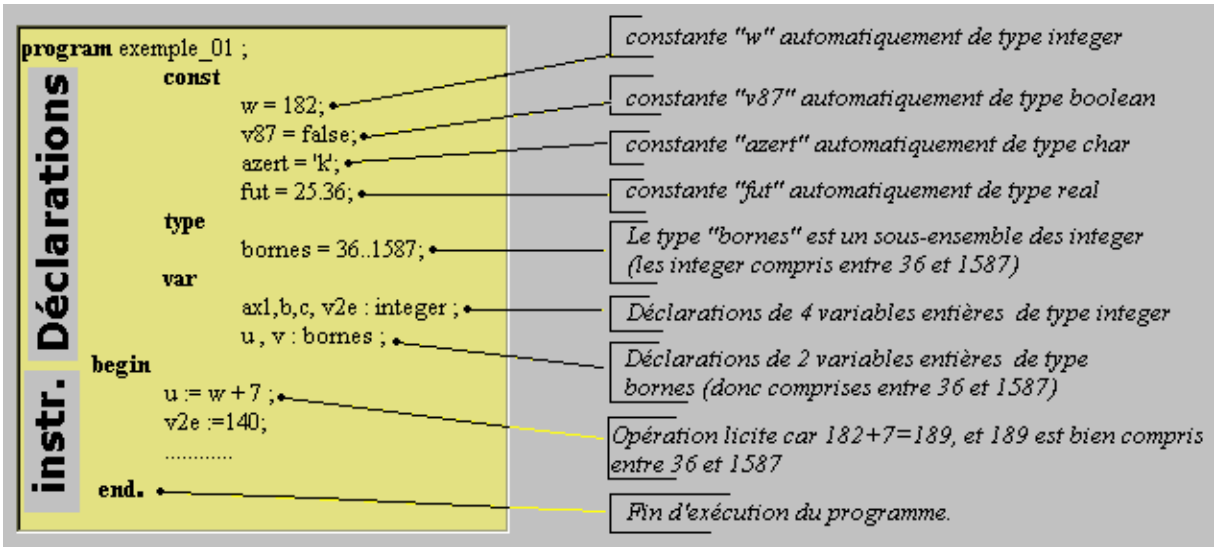
Bloc :



En pratique un bloc Pascal contient deux parties : des déclarations et des instructions



Exemple de programme *avec Bloc :*



2. Les opérateurs en pascal

Ce sont les règles de composition qui précisent la priorité retenue entre les différents opérateurs du langage. Ces priorités sont réparties en **4** niveaux :

- plus haut niveau de priorité 4 : opérateur unaire **not**
- niveau de priorité 3 : opérateurs multiplicatifs (***, /, div, mod, and**)
- niveau de priorité 2 : opérateurs additifs (**+ , - , or**)
- plus bas niveau de priorité 1 : opérateurs relationnels (< , >, etc…)

2.1 Liste de tous les opérateurs selon le type de données en Pascal.

integer x integer ⟶ integer

op	signification	exemple
*	multiplication	2 * (x-8)+a*b
div	division euclidienne	u div (x*8)+a div b
+	addition	x - (x+8)+a*b
-	soustraction	x - (x-8)-a-b
mod	reste euclidien	

opérateurs sur les entiers

real x real ⟶ real

op	signification	exemple
*	multiplication	2 * (x-8)+a*b
/	division	u / (x*8)+a/b
+	addition	x - (x+8)+a*b
-	soustraction	x - (x-8)-a-b

opérateurs sur les réels

bolean x boolean ⟶ boolean

op	signification	exemple
or	ou	(a or b) or ((c or d))
and	et	(a and b) and ((c and a or d))
not	non	not a or not(b and c)

opérateurs sur les booléens

set of x set of → set of

op	signification	exemple
*	intersection	A * B
+	union	A+B
-	différence	A-B

set of x set of → boolean

op	signification	exemple
=	égalité	A =B
<>	différent de	A<>B
<=	inclusion	A <= B

T x set of → boolean

op	signification	exemple
in	appartient à	x in E

opérateurs sur les ensembles : set of

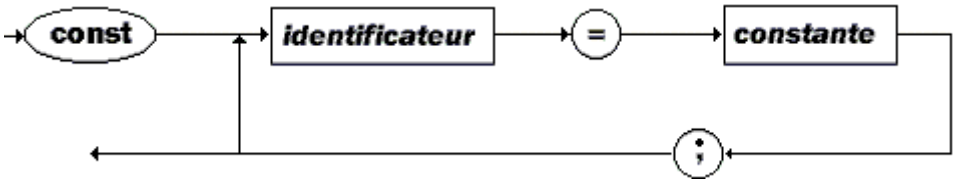
T x T ⟶ boolean

op	signification	exemple
<	inférieur strict	'b' < 'k' ; -8 < 25
>	supérieur strict	'p' > 'k' ; 98 > 32
=	égalité	x = 3 ; a = b
<=	inférieur large	'b' <= 'k' ; -8 <= 25
>=	supérieur large	'p' >= 'k' ; 98 >= 32
<>	différent	'g'<>'m' ; 0 <>x

opérateurs de comparaison sur un type T

2.4 Déclarations des constantes

Sert à associer un identificateur à une valeur de constante, sa valeur est non modifiable dans le reste du programme. Il existe 3 identificateurs de constantes prédéfinis : **True** , **False** , et **Nil** .



Exemple :
program exemple_03 ;
const
x = 12; <---------- *x est une constante de type* **integer**.
a2 = true; <---------- *a2 est une constante de type* **boolean**.
Y = 'h'; <---------- *Y est une constante de type* **char**.
r2 = 25.36; <---------- *r2 est une constante de type* **real**.

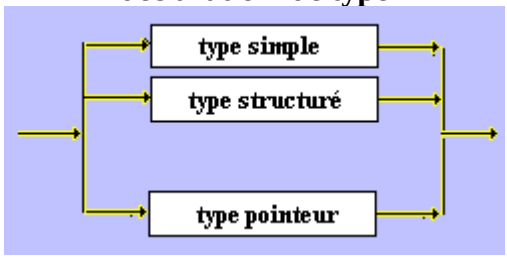
3. Déclarations des types en pascal

Les types sont utilisés pour créer de nouveaux domaines de définition de variables. Une déclaration d'un nouveau type de données sert à associer un identificateur à un type de données construit par l'utilisateur.

Cette construction est élaborée à l'aide de constructeurs de type et détermine l'ensemble des valeurs possibles des variables du nouveau type.

On classe les types en 3 catégories :

< déclaration de type >

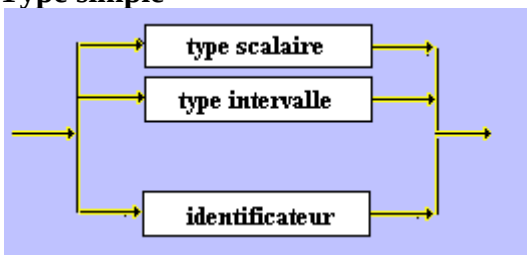


3.1 Déclarations des types simples

Cette déclaration est composée des :

- type scalaire
- type intervalle
- identificateur d'un type déjà déclaré

< Type simple >



< Les types scalaires > (ils sont de 2 sortes) :

Les types prédéfinis :

- **integer**
- **real**
- **char**
- **boolean**
- **string**

Les types énumérés :

identif0 = **(** identif1,identif2,....,identifk **)**

3.2 Déclarations des types énumérés

Type identif0 = **(** identif1,identif2,....,identifk **) ;**

Il s'agit ici d'une définition en extension des éléments du type. Les *identifn* sont des constantes symboliques de base du type et doivent être tous différents dans la même énumération, et ne peuvent se retrouver ni dans une autre énumération, ni redéfinis ailleurs.

> Ce type est doté d'une fonction spécifique : **ord** qui dénote le numéro d'ordre d'un élément dans l'ensemble des valeurs du type (attention l'ordre est construit de gauche à droite et la numérotation débute à la valeur 0).

Exemple : **créons un type jour de la semaine**

Type
jour = **(** lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche **) ;**

Le type énuméré jour voit ses constantes de base automatiquement numérotées de 0 à 6 comme l'indique le tableau ci-après:

lundi	mardi	mercredi	jeudi	vendredi	samedi	dimanche
0	1	2	3	4	5	6

Ainsi le rang est accédé par la fonction ***ord*** *:*

ord(jeudi) = 3
ord(lundi) = 0

Remarque *:*

Les types scalaires sauf le type real bénéficient de 2 fonctions **succ** et **pred**

succ : T $\rightarrow$ T / **succ** (a_i) = a_{i+1} (successeur dans T , lorsqu'il existe)
pred : T $\rightarrow$ T / **pred** (a_i) = a_{i-1} (prédécesseur dans T, lorsqu'il existe)

3.3 Déclarations des types intervalles

Il peut être défini comme un intervalle fermé borné d'un autre type scalaire, sauf **real**. Les constantes représentent les bornes de l'intervalles (la constante de gauche représente la borne inférieure, la constante de droite représente la borne supérieure)

Exemples :

```
Type
   jour = (lundi , mardi , mercredi , jeudi , vendredi , samedi , dimanche ) ;

   mois = 1..12 ;  // intervalle sur les entiers compris entre 1 et 12
   week_end = vendredi..dimanche ; // intervalle sur les jour : vendredi , samedi , dimanche
   lettre_min = 'a'..'z' ; // intervalle sur les caractères de type lettres minuscules
   lettre_maj = 'A'..'Z' ; // intervalle sur les caractères de type lettres majuscules
```

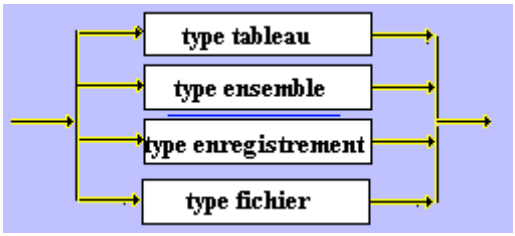
Il est bien entendu possible de déclarer ensuite des variables sur ces types :

```
   Var
     x : mois ;
     y : week_end
     z : lettre_maj
```

3.5 Déclarations des types structurés

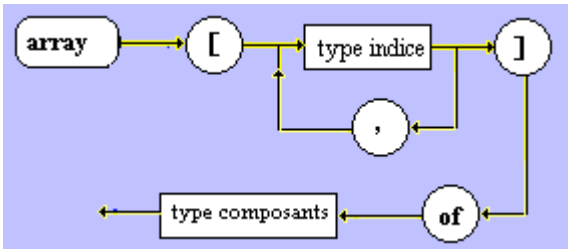
Il est donc possible en Pascal de construire et d'utiliser des variables de type simple comme integer, real, boolean, string, char, énumérés et intervalles. Mais il est aussi possible de travailler avec des familles de variables de même type ayant une structuration spécifique ou bien avec des structures contenant des variables ayant des types différents. Ces familles sont appelées des types structurés.

Elles sont au nombre de 4 en pascal :



Une définition de type structuré, précise par l'intermédiaire du constructeur de type, la méthode de structuration et le type des données le composant.

3.6 Déclarations de type tableau



Le type tableau est défini par le constructeur de type **array[ ] of.**

C'est une structure homogène, formée d'un nombre fixe de composants qui sont tous du même type de base. Tous les composants d'un tableau sont désignés par des indices, qui sont des expressions appartenant au **type indice** du tableau.

> Un tableau est en fait une structure de donnée à *accès aléatoire* , c'est à dire que tous ses composants peuvent être sélectionnés et atteints de manière égale. Ils sont rangés dans l'ordre des indices.

Un tableau à n dimensions (un vecteur est représenté par un tableau à une dimension, une matrice par un tableau à deux dimensions...) est défini par n **types d'indices** séparés par des virgules.

Un **type indice** est un type simple sauf **real** et **integer**.

Exemple :

```
Type
  jour = ( lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche ) ;
  mois = 1..12 ;
  week_end = vendredi..dimanche ;
  lettre_min = 'a'..'z' ;
  lettre_maj = 'A'..'Z' ;
  tableau_01 = array[jour] of mois;
  tableau_02 = array[jour] of array[1..30] of mois;
  tableau_03 = array[jour,1..30] of mois;
  tableau_04 = array[lettr_min,0..5,jour,boolean] of char;
var
  T1 : tableau_01;
  T2 : tableau_02;
  T3 : tableau_03;
  T4 : tableau_04;
  etc.....
```

ATTENTION :

> *Notons ici que malgré la similitude de construction des deux types tableau_02 et tableau_03 (ce sont des types de matrices où l'indice ligne varie dans le type jour, et l'indice colonne varie dans le type 1..30), ce ne sont pas des types identiques, car ils sont déclarés séparément.*
>
> Donc dans l'exemple précédent, T2 et T3 **ne** sont **pas** des tableaux du même type.

Accès aux variables d'un type tableau

Il faut, afin de pouvoir accéder à un composant d'un tableau, utiliser des indices obligatoirement de même type et en même nombre qu'indiqués dans la déclaration.

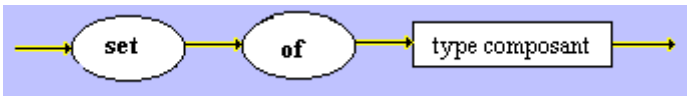
Exemple : (en reprenant les déclarations précédentes)

```
var
   T1 : tableau_01;
   T2 : tableau_02;
   T3 : tableau_03;
   T4 : tableau_04;
   m : mois;
   j : jour;
   k : 1..30;
   L,b: boolean;
   n : integer;
   c : lettre_min;
```

Les écritures suivantes sont licites :

```
 j:= jeudi; k:= 20; c:='f'; L:=false; b:=true; n:=2;
T1[mardi]:= 8; T1[j]:= 10;
T2[mardi,5]:= 8; T2[mardi] [5]:= 8; T2[j,k-3]:= 8; T2[j] [k-3]:= 8;
T3[mardi,5]:= 8; T3[mardi] [5]:= 8; T3[j,k]:= 8; T3[j] [k]:= 8;
T4['t',3,samedi,true]:= 'h'; T4['t'][3][samedi][true]:= 'h';
T4[c,n+2,j,L or b]:= '+'; ..... etc
```

3.7 Déclarations de type ensemble



Un type ensemble est défini d'un manière extensive par le constructeur **set of**, le domaine des valeurs de ses éléments par son type de base.

le t**ype ensemble** est un type simple sauf **real** et **integer**.

C'est un ensemble fini et l'on peut construire tous ses sous-ensembles :

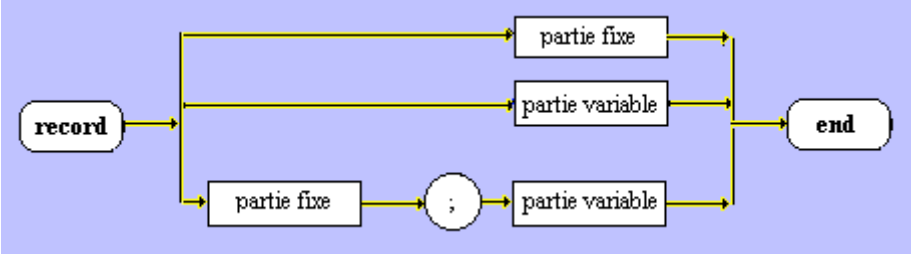
Exemple :

```
Type
   couleur = (noir,blanc);
   ens_couleur = set of couleur;
var
   x,y,z,t : ens_couleur;
```

```
begin
   x := [ ] ; <--- ensemble vide (0 élément)
   y := [noir]; <--- ensemble (1 élément)
   z := [blanc]; <--- ensemble (1 élément)
   t := [noir,blanc]; <--- ensemble (2 éléments : maximum possible de l'exemple)
etc.....
```

On peut dire en fait que le type ens_couleur est l'ensemble P(couleur) (ensemble des parties) et que toute variable du type ens_couleur est un sous-ensemble de P(couleur).

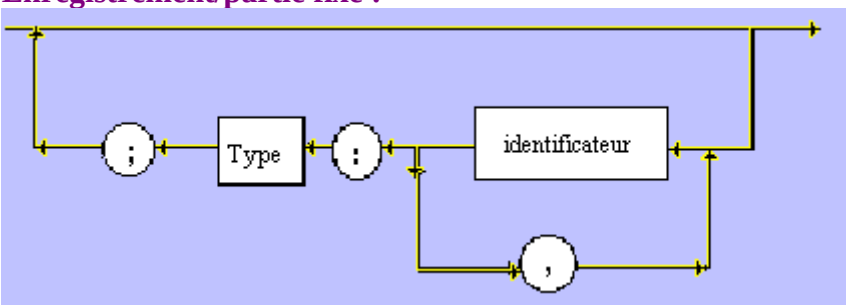
3.7 Déclarations de type enregistrement



Le type enregistrement est une collection de composants appelés **champs** de l'enregistrement. Ils peuvent être d'un type quelconque sauf le type fichier. C'est une structure hétérogène.

> Tous les identificateurs de champs d'une même structure enregistrement doivent être différents à l'intérieur de l'enregistrement. Ils permettent d'accéder directement aux éléments de l'enregistrement.

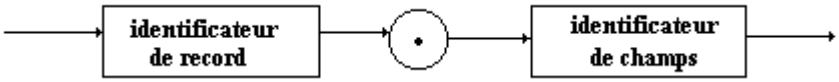
Enregistrement/partie fixe :



Exemple:

```
Type
   enregis = record
      jour : (lundi,mardi,dimanche);
      x,y : integer;
      mois : 1..12;
      T_paie : array[boolean,1..30] of real;
   end;
```

Enregistrement/accès aux champs :



L'accès aux champs à l'intérieur d'un enregistrement s'effectue à l'aide de l'identificateur de l'enregistrement (**identif de record**), puis de celui du champs (**identif de champs**) auquel on désire accéder, dans cet ordre, comme en désignant un chemin accédant aux éléments en écrivant de gauche à droite.

Exemple :

```
Type
  Tenregis = record
      jour : (lundi,mardi,dimanche);
      x,y : integer;
      mois : 1..12;
      T_paie : array[boolean,1..31] of real;
  end;
var
  A : Tenregis;
begin
  A.jour:=;mardi;
  A.mois:=8;
  A.y:=125;
  A.x:=0;
  A.T_paie[false,A.mois] := -2.37
  etc.....
```

4. Instructions en pascal

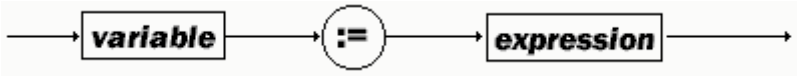
Ce sont les traductions des instructions algorithmiques de notre langage de description formelle d'algorithme que nous avons dénommé LDFA.

LDFA	Pascal
Ω (instruction vide)	pas de traduction
debut i1 ; i2; i3; ..... ; ik **fin**	**begin** i1 ; i2; i3; ..... ; ik **end**
x ← a	x := a
;	(ordre d'exécution) ;
Si P **alors** E1 **sinon** E2 **Fsi**	**if** P **then** E1 **else** E2 *(attention défaut, pas de fermeture !)*
Tantque P **faire** E **Ftant**	**while** P **do** E *(attention, pas de fermeture)*

répeter E **jusquà** P	**repeat** E **until** P
lire (x1,x2,x3.....,xn)	read(fichier,x1,x2,x3.....,xn) readln(x1,x2,x3.....,xn) Get(fichier)
ecrire (x1,x2,x3.....,xn)	write(fichier,x1,x2,x3.....,xn) writeln(x1,x2,x3.....,xn) Put(fichier)
pour x ← a **jusquà** b **faire** E **Fpour**	**for** x:=a **to** b **do** E *(croissant)* **for** x:=a **downto** b **do** E *(décroissant)* *(attention, pas de fermeture)*
SortirSi P	**if** P **then** Break

4.1 Instruction d'affectation

L'affectation est applicable à tous les genres de variables du pascal sauf au type **file of**.



Sémantique:

- Evaluation de la partie droite (l'expression)
- Transfert de la valeur calculée dans la partie gauche (la variable)

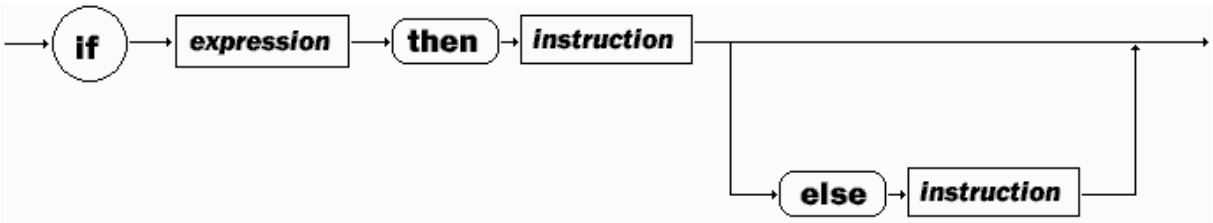
Exemple :

```
program Affectation ;
type
   Temperature = -20 .. 40 ;
   LettreMin = ' a ' .. ' z ' ;
   Jour = ( lundi , mardi , mercredi , jeudi ) ;
var
    a : integer ;  b : char ;
    c : string ;
    Temp : Temperature ; Lmin :  LettreMin ;
    Day : Jour ;
begin
   Temp := 18 ;
    a := (2+Temp)*4 ;
    b := 'F' ;
    c := 'bon'+'jour' ;
    Lmin := 'f' ;
    Day := mercredi ;
 end.
```

Après affectations :

Temp *vaut* 18
a *vaut* 80
b *vaut* 'F'
c *vaut* 'bonjour'
Lmin *vaut* 'f'
Day *vaut* mercredi

4.2 Instruction de condition



Dans l'instruction **if**, l'expression est un prédicat (expression contenant des variables, prenant la valeur vrai ou faux), les blocs <***instruction***> représentent soit une instruction simple, soit une instruction composée (**begin** ..... **end**).

Sémantique:

cas du ***if...then***

- Si l'expression est vraie, le bloc d'instruction situé après le **then** est exécuté et le ***if...then*** s'arrête
- Si l'expression est fausse le ***if...then*** s'arrête.

cas du ***if...then...else***

- Si l'expression est vraie, le bloc d'instruction situé après le **then** est exécuté et le ***if...then...else*** s'arrête.
- Si l'expression est fausse, le bloc d'instruction situé après le **else** est exécuté et le ***if...then...else*** s'arrête.

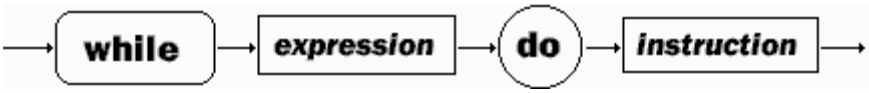
Exemple :

```
program Condition ;
var
    x, y ,z : integer ;

begin
   x := 10 ;
   y := x*4 ;
   if y>100 then z := y
   else z := 0;
   if z = 0 then
      y := 0
   x := 0 ;
end.
```

Exécution pas à pas :

x *vaut* 10
y *vaut* 40
y>100 *est* **false**
donc **z** *vaut* 0
z=100 *est* **true**
donc y *vaut* 0
x *vaut* 0
(à la fin : x=0, y=0, z=0)

4.3 Instruction d'itération while…do



Dans l'instruction **while…do**, l'expression est un prédicat (expression contenant des variables, prenant la valeur vrai ou faux), le blocs <***instruction***> représente soit une instruction simple, soit une instruction composée (**begin** ..... **end**).

Sémantique:

C'est une instruction de boucle.

- Tant que l'expression reste vraie, le bloc d'instruction est réexécuté.
- Dès que l'expression est fausse le ***while…do*** s'arrête.

C'est une boucle non finie (c-à-dire que l'on ne peut pas connaître dans les cas de figure si une boucle quelconque de ce type s'arrêtera après un nombre fini d'exécution).

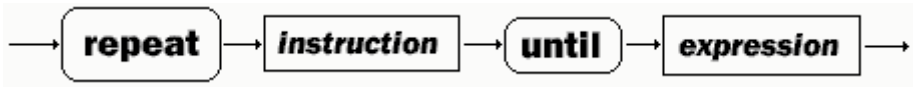
Exemple :

```
program WhileDo ;
var
    x, y  : integer ;

begin
   x := 1 ;
   y := 0 ;
   while x<4 do
   begin
      x := x+1 ;
      y := y +x
   end;
   writeln ('x=', x , 'y=', y)
end.
```

Le programme écrit :
x=4 y=9

Exécution pas à pas :
x *vaut* 1
y *vaut* 0
x<4 *est* **true**
donc **x** *vaut* x+1 *soit* 2
et **y** *vaut* y+x soit 2
x<4 *est* **true**
donc **x** *vaut* x+1 *soit* 3
et **y** *vaut* y+x *soit* 5
x<4 *est* **true**
donc **x** *vaut* x+1 *soit* 4
et **y** *vaut* y+x *soit* 9
x<4 *est* **false** donc **arrêt**
(à la fin : x=4, y=9)

4.4 Instruction d'itération repeat…until



Dans l'instruction **repeat…until**, l'expression est un prédicat (expression contenant des variables, prenant la valeur vrai ou faux), le blocs <***instruction***> représente soit une suite d'instructions simples.

Sémantique:

C'est une instruction de boucle.

- Tant que l'expression reste fausse, le bloc d'instruction est réexécuté.
- Dès que l'expression est vraie le ***repeat…until*** s'arrête.

C'est une boucle non finie (c-à-dire que l'on ne peut pas connaître dans les cas de figure si une boucle quelconque de ce type s'arrêtera après un nombre fini d'exécution).

La différence avec le **while .. do** réside dans le fait que le **repeat ... until** exécute toujours au moins une fois le bloc d'instructions avant d'évaluer l'expression booléenne alors que le **while ... do** évalue immédiatement son expression booléenne avant d'exécuter le bloc d'instructions.

Exemple :

```
program RepeatUntil ;
var
    x, y  : integer ;

begin
   x := 1 ;
   y := 0 ;
   repeat
     x := x+1 ;
     y := y +x
   until x>=4;
   writeln ('x=', x , 'y=', y)
end.
```

Le programme écrit :

x=4 y=9

Exécution pas à pas :

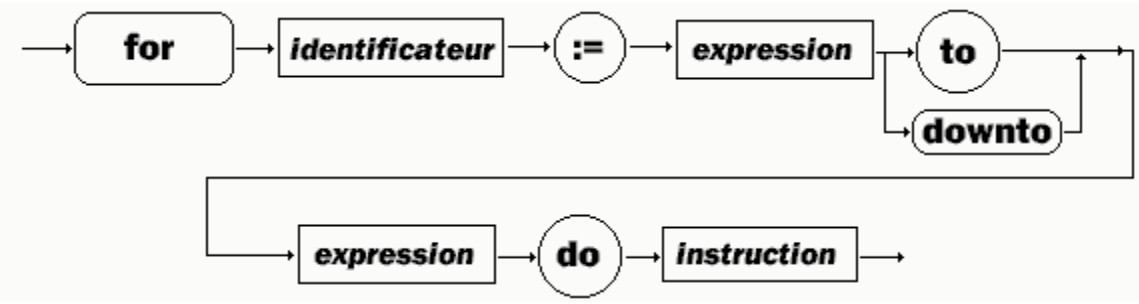
x *vaut* 1
y *vaut* 0
on entre dans le repeat
donc **x** *vaut* x+1 *soit* 2
et **y** *vaut* y+x soit 2
x>=4 *est* **false**
donc **x** *vaut* x+1 *soit* 3
et **y** *vaut* y+x *soit* 5
x>=4 *est* **false**
donc **x** *vaut* x+1 *soit* 4
et **y** *vaut* y+x *soit* 9
x>=4 *est* **true** donc **arrêt**
(à la fin : x=4, y=9)

Ce programme fournit le même résultat que celui de la boucle **while…do**, car il y a une correspondance sémantique entre ces deux boucles :

repeat <instruction> **until** <expr>	<instruction> **;** **while not**<expr> **do** <instruction>

4.5 Instruction d'itération for…do

C'est une instruction de boucle, il y a deux genres d'instructions **for (for...to** et **for...downto) :**



Version **for** <***identificateur***> := <***Expr1***>**to** <***Expr2***> **do** <***Instruction***> :

- ***identificateur*** est une variable qui se dénomme indice de boucle.
- <***Expr1***> et <***Expr2***> sont obligatoirement des expressions du même type que la variable d'indice de boucle ***identificateur***.
- < ***Instruction*** > est un bloc d'instruction simple ou composée (**begin** ..... **end**).

Version **for** <***identificateur***> := <***Expr1***> **downto** <***Expr2***> **do** <***Instruction***> :

- même signification des constituants que pour la version précédente, seul le sens de parcours diffère (par valeurs croissantes pour un **for...to**, par valeurs décroissantes pour un **for...downto**).

Sémantique:

L'indice de boucle prend toutes les valeurs (par ordre croissant ou décroissant selon le genre de **for**) comprises entre <***Expr1***> et <***Expr2***> bornes inclues.

Tant que la valeur de l'indice de boucle ne dépasse pas

- par valeur supérieure dans le cas du **for...to**,
- ou par valeur inférieure dans le cas du **for...downto**

la valeur de <***Expr2***>, le bloc d'instruction est réexécuté.

C'est une boucle **finie** (c-à-dire que l'on connaît à l'avance le nombre de tours de boucle).

Exemple :

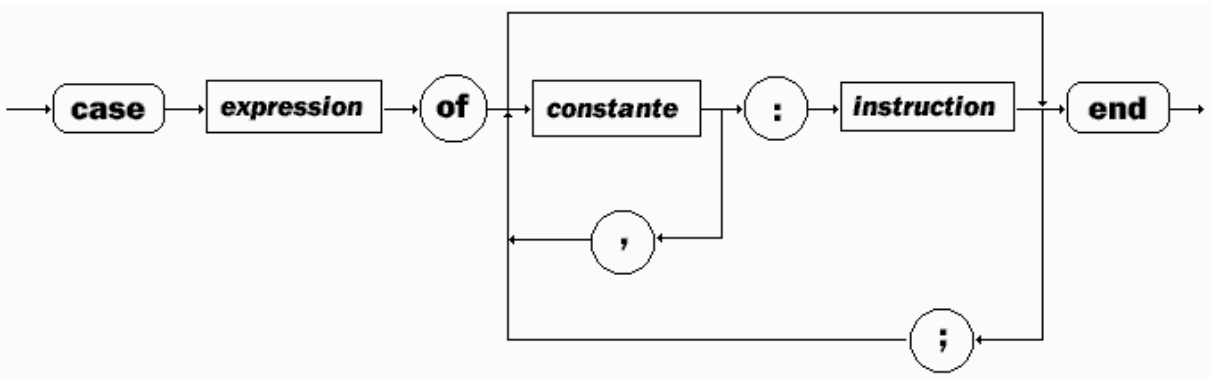
```
program ForDo ;
var    x, y  : integer ;
   begin
     y :=0 ;
     for x := 1 to 3 do
          y := y +x
   end.
```

Exécution de chaque tour de boucle :

y *vaut* 0
x *vaut* 1 => **y** *vaut* 0+1=1
x *vaut* 2 => **y** *vaut* 1+2=3
x *vaut* 3 => **y** *vaut* 3+3=6
x *vaut* 4 => **arrêt**
(à la fin : x=4, y=6)

4.6 Instruction case…of

C'est une instruction de choix



<***expression***> doit être de l'un des types : integer, char, boolean, énuméré, intervalle .
<***constante***> doit obligatoirement être du même type que <***expression***>
<***Instruction***> est un bloc d'instruction simple ou composée (**begin** ..... **end**).

Sémantique:

C'est une instruction structurée équivalente à une série de **if**...**then**...**else** imbriqués. Cette instruction lorsque cela est possible, doit être préférée à un emboîtement de **if**...**then**...**else** dont la lisibilité n'est en fait pas optimale.

if...**then**...**else** imbriqués	**case ... of** équivalent
if x = 3 **then** E1 **else** **if** x = 4 **then** E2 **else** **if** x = 5 **then** E2 **else** **if** x = 6 **then** E2 **else** **if** x = -5 **then** E3 **else** Ef	**case** x **of** 3 : E1 ; 4..6 : E2 ; -5 : E3 ; **else** Ef **end**

Exemple :

```
program CaseOf ;
var    x, y  : integer ;
   begin
     y :=1 ;
     for x := 0 to 4 do
        case x+1 of
          0..3 : y :=y*2 ;
          4 : y := y+100
          else y:=0 ;
        end
   end.
```

y *vaut* 1

Exécution du case dans la boucle :

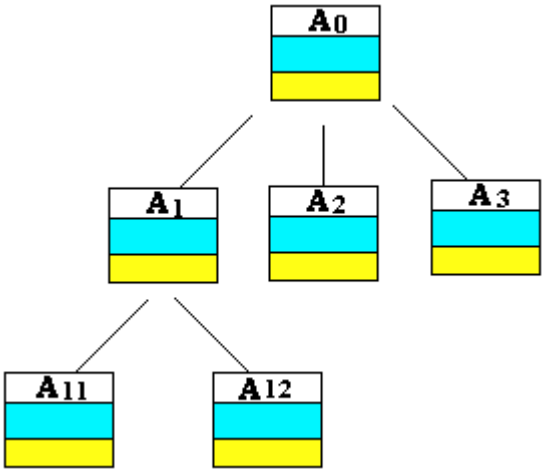
x *vaut* 0 => **x+1** *vaut* 1 (dans 0..3) => **y** *vaut* 1*2=2
x *vaut* 1 => **x+1** *vaut* 2 (dans 0..3) => **y** *vaut* 2*2=4
x *vaut* 2 => **x+1** *vaut* 3 (dans 0..3) => **y** *vaut* 4*2=8
x *vaut* 3 => **x+1** *vaut* 4 => **y** *vaut* 8+100=108
x *vaut* 4 => **x+1** *vaut* 5 (else) => **y** *vaut* 0
x *vaut* 5 => **arrêt**

(à la fin : x=4, y=0)

5. Fonctions et procédures en pascal

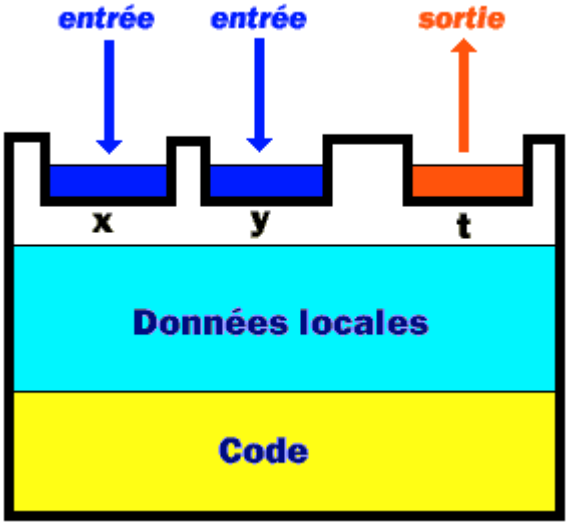
Le langage Pascal a été conçu à l'origine comme un langage pédagogique d'implantation de la programmation de type algorithmique; grâce à son extension objet Delphi il est utilisé comme outil de développement professionnel en entreprise.

La programmation algorithmique est une programmation hiérarchisée descendante.

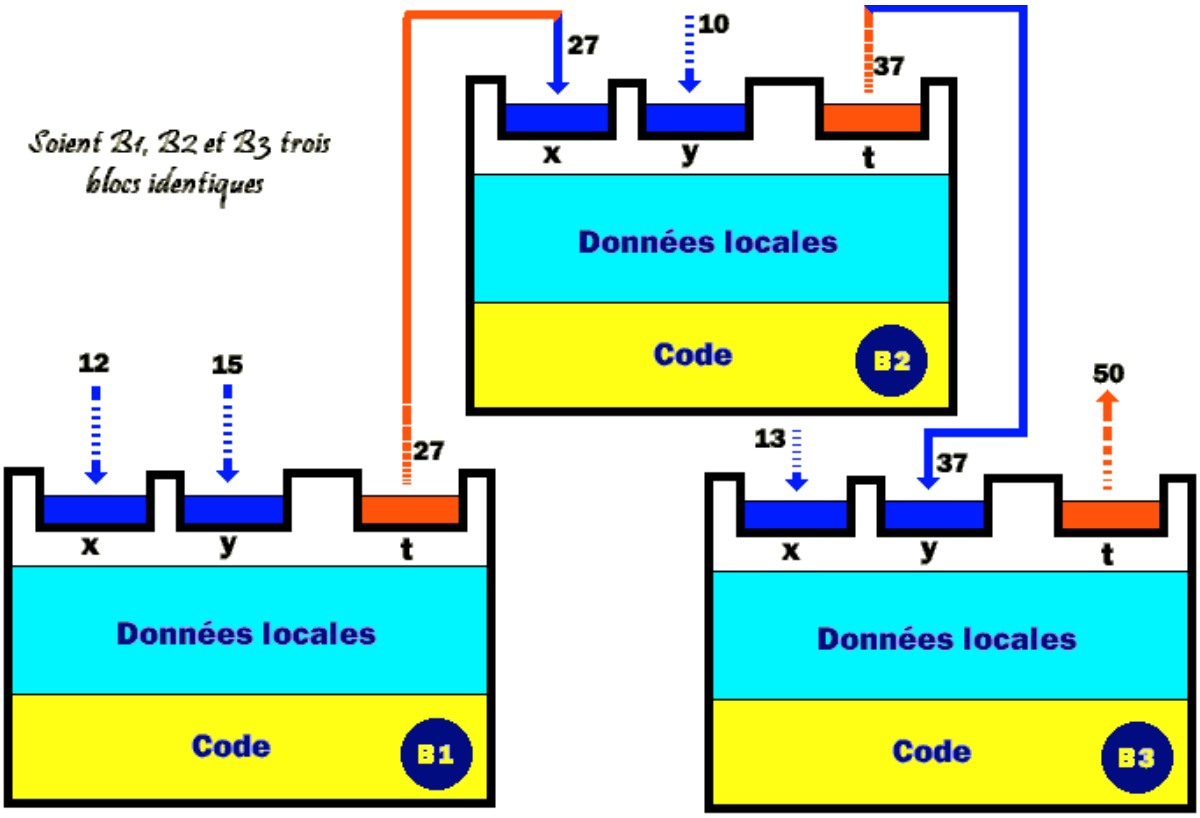


Cette décomposition descendante hiérarchique est construite à l'aide de blocs de programme notés aussi des sous-programmes.

Un bloc comporte donc des données locales, du code (instructions ou corps du bloc), des données d'entrée et/ou des données de sortie (permettant les échanges d'informations entre les différents blocs de la hiérarchie) :



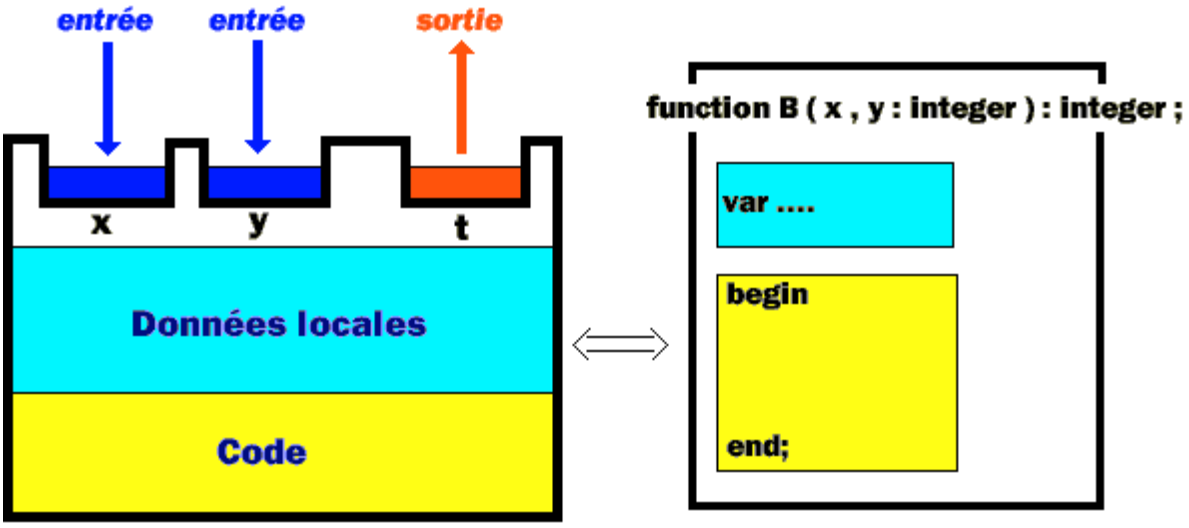
L'exemple ci-après représente trois blocs B1, B2 et B3 échangeant des informations (en fait chacun calcule la somme des deux entiers qu'il reçoit en entrée et renvoie leur somme :



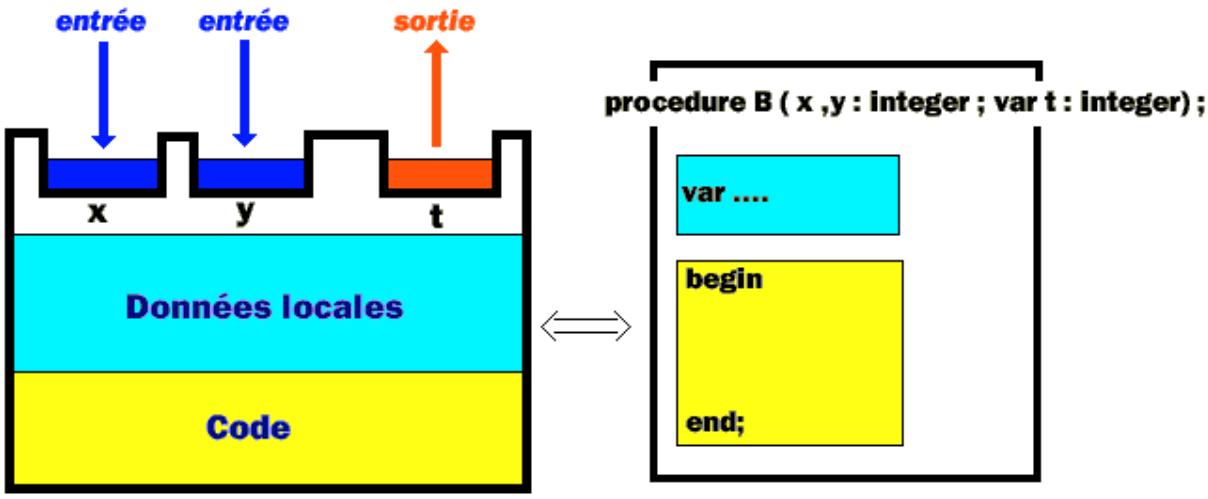
Le bloc B1 reçoit en entrée 12 et 15 et renvoie la somme 12+15 = 27 vers le bloc B2, la valeur 27 devient une donnée d'entrée pour le bloc B2 qui reçoit comme autre entrée la valeur 10. Le bloc B2 renvoie vers le bloc B3 le résultat 27+10 = 37 etc...

Nous remarquons que chaque bloc est indépendant des autres blocs. La seule liaison qui intervienne ici se situe dans le passage des données d'un bloc vers un autre bloc. Le code et les données locales d'un bloc fixé sont inaccessibles aux autres blocs.

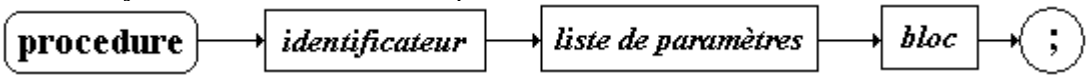
En pascal les blocs sont implémentés soit par des fonctions :



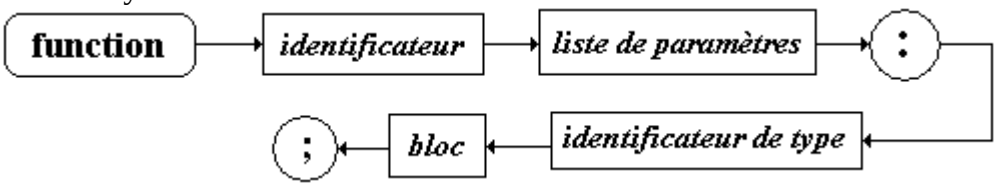
En pascal les blocs sont implémentés aussi par des procédures :



Voici la syntaxe de déclaration des procédures en Pascal :



Voici la syntaxe de déclaration des fonctions en Pascal :



- <***identificateur***> est le nom de la procédure ou de la fonction (choisi par vous)
- <***liste de paramètres***> est soit vide, soit elles contient entre parenthèses et séparés par des point-virgules la liste des paramètres formels.
- <***bloc***> est une instruction composée (**begin** ..... **end**).
- <***identificateur de type***>, dans le cas d'une fonction représente le type du résultat renvoyé par la fonction.

Exemples de déclarations avec et sans paramètres formels :

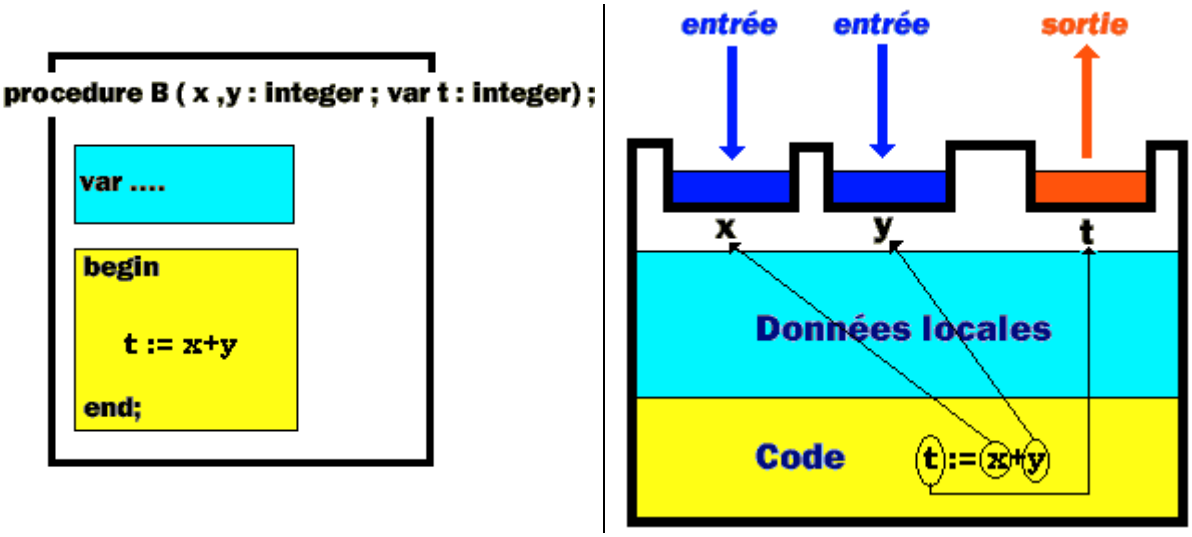
procedure Somme (x,y :integer; **var** z :integer) ;
begin
z := x +y
end ;
	function Somme (x,y :integer): integer ;
	begin
	result := x +y
	end ;

procedure Somme ;
var x, y : integer ;
begin
y :=1 ; x := 2;
writeln(x+y)
end ;
	function Somme : integer ;
	var x, y : integer ;
	begin
	y :=1 ; x := 2;
	result := x +y
	end ;

6. Paramètres en pascal

On s'intéresse dans ce paragraphe aux rapports qu'il y a entre un programme appelant et un sous-programme appelé uniquement en Pascal.

Soit par exemple une procédure B ayant 3 paramètres formels et renvoyant dans le troisième paramètre la somme des deux premiers :



Les paramètres formels d'une procédure jouent le rôle de variables muettes et servent à décrire le fonctionnement d'une procédure. Ils ont la même utilisation qu'une variable dans un polynôme mathématique. Les deux écritures $P(x) = 3x^2 - 4x + 5$ et $P(t) = 3t^2 - 4t + 5$ représentent mathématiquement le même polynôme, il en est de même pour une procédure.

on peut changer tous les paramètres formels d'une procédure sans en changer son fonctionnement

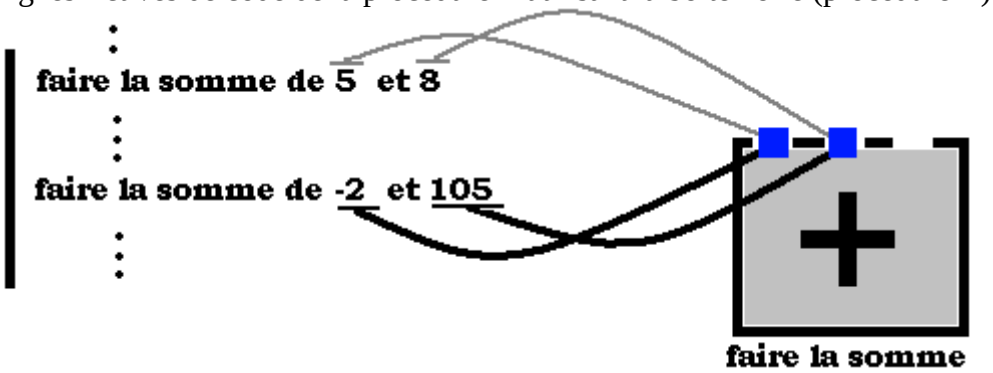
Les deux déclarations ci-dessous sont identiques :

```
procedure B (x , y :integer; var t :integer) ;
begin
  t := x +y
end ;

procedure B (a , b integer; var c :integer) ;
begin
  c := a +b
end ;
```

L'intérêt pratique d'une procédure et en général d'un sous-programme est essentiellement de pouvoir exécuter toujours la même action mais avec des valeurs différentes.

Par exemple une procédure P qui utilise un autre procédure B qui fait la somme, de deux entiers. La procédure B fonctionne comme une sorte de boîte noire qui reçoit deux valeurs en entrée et qui retourne leur somme comme dans le pseudo-code ci-dessous :

Lignes fictives de code de la procédure P utilisant la boîte noire (procédure B)



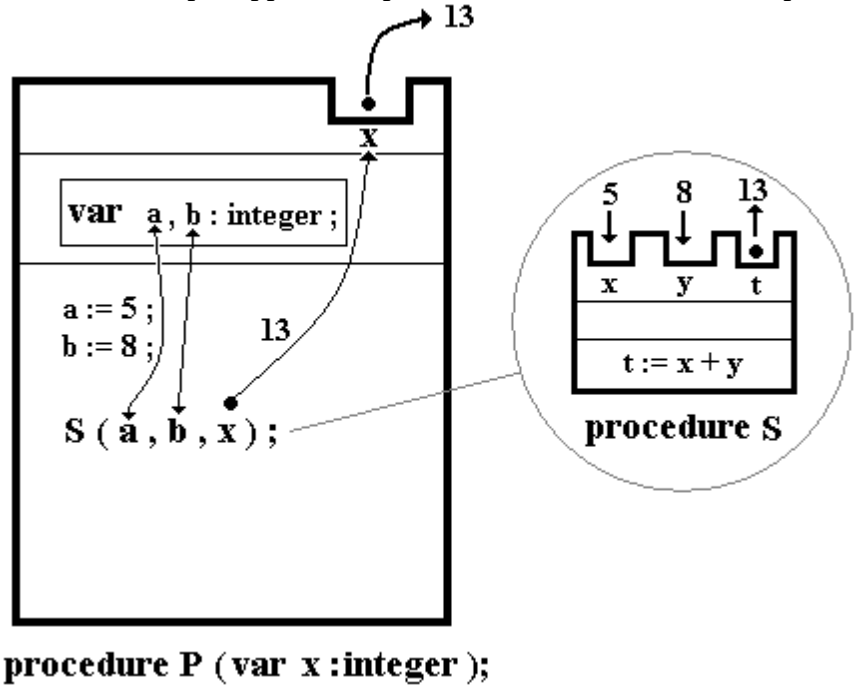
La boîte "faire la somme" est utilisée une première fois pour sommer 5 et 8, puis plus loin elle est utilisée une deuxième fois pour sommer -2 et 105.

Le mécanisme qui permet d'utiliser la procédure B dans le code de la procédure P se dénomme l'**appel** de procédure. P se dénomme la procédure **appelante**.

Reprenons l'exemple du polynôme écritures $P(x) = 3x^2 - 4x + 5$, nous savons qu'en donnant une valeur effective à la variable x (par exemple x = 2) on obtient un résultat noté P(2) qui vaut: $P(x) = 3.2^2 - 4.2 + 5 = 9$.

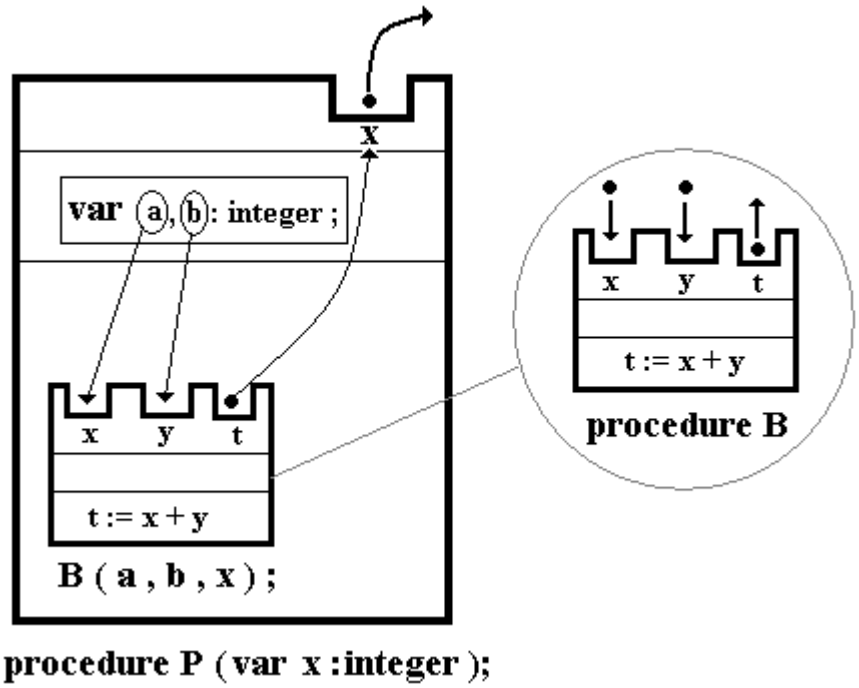
L'appel de procédure est un procédé très semblable au calcul du polynôme sur une valeur. La procédure a besoin qu'on lui fournisse des variables contenant effectivement des valeurs. De telles variables se dénomment les **paramètres effectifs** de la procédure.

Précisons un peu plus l'utilisation d'une procédure S avec des variables. Supposons que S serve à calculer la somme de deux valeurs **5** et **8** contenues respectivement dans deux variables locales **a** et **b** d'une autre procédure nommée P dont le seul paramètre **x** renvoie le résultat **13** du calcul obtenu par appel de la procédure B dans le code de la procédure P :



L'appel S(a,b,x) s'effectue sur des paramètres effectifs qui sont nécessairement des variables existantes, soit déclarées dans un paragraphe var dans la zone des données locales, soit déclarées en tant que paramètres de la procédure appelante.

L'appel se fait avec un nombre de paramètres effectifs égal à celui des paramètres formels en respectant l'ordre et la cohérence des types. On peut imaginer que lors d'un appel à la procédure S par le code de la procédure S, le code de la procédure S vient s'imbriquer *fictivement* dans le code de P à l'endroit de l'appel avec comme variables les paramètres effectifs :



Comment a lieu cet appel, cette inclusion fictive du code ?

On dénomme l'action qui consiste à appeler sur des paramètres effectifs, le **passage** des paramètres effectifs ou encore la **transmission** des paramètres effectifs.

Il faut savoir qu'un **paramètre effectif** transmis au sous-programme appelé est un **moyen d'utiliser ou d'accéder** à une information appartenant au bloc appelant (le bloc appelé peut être le même que le bloc appelant, il s'agit alors de récursivité).

Pascal ne dispose que de 2 modes de passage sur les 5 modes généraux :

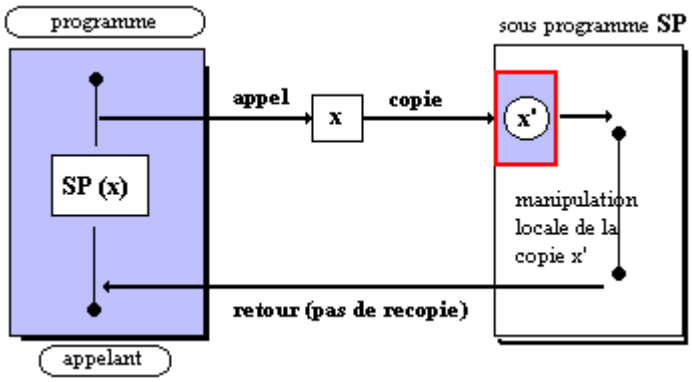
- Le passage par **valeur,**
- le passage par **référence** ou **adresse.**

6.1 Lecture seulement : passage par valeur

Dans un passage par valeur, le paramètre formel est considéré comme une variable locale dans le corps du sous-programme. Sa valeur est initialisée au début de chaque exécution du sous-programme avec la valeur du paramètre effectif correspondant.

Il y a recopie de la valeur du paramètre effectif dans une zone spécifique locale à la procédure. Toutes les opérations qui sont effectuées sur le paramètre formel n'affectent que cette valeur locale.

Ecriture en Pascal **procedure** sp(... x: **real** ....)

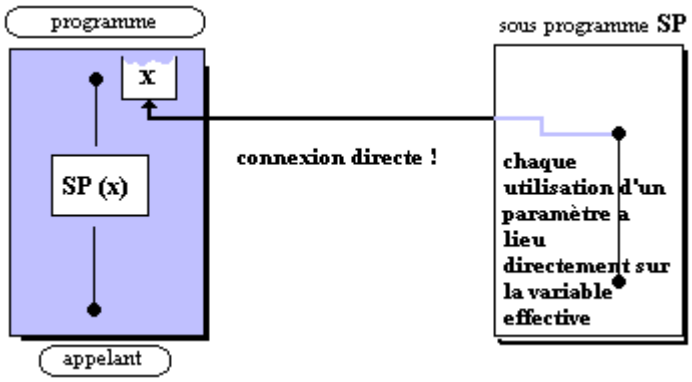


passage par valeur

6.2 Accès direct : passage par adresse ou par référence

Dans un passage par adresse le paramètre formel est traité comme une variable dont l'adresse qui est transmise au moment de chaque appel, est celle du **paramètre effectif** correspondant. L'adresse de la variable effective autorise toutes les modifications immédiatement sur cette variable quelle que soit sa localisation.

Ecriture en Pascal **procedure** sp (....**var** x : **real** .....)



passage par référence

Comparaison des avantages et des inconvénients des 2 modes

Passage par valeur :

- **Avantage** : sécurité et protection des informations.
- **Inconvénient** : lenteur due à la recopie des données et doublement de la place mémoire occupée (mais convient bien pour des variables simples !).

Passage par référence :

- **Avantage** : rapidité d'accès aux données, moindre occupation mémoire puisqu'il ne s'agit que d'une adresse.
- **Inconvénient** : ce mode est dangereux à cause de la non protection des données et de la nécessité qu'il y a de connaître la façon dont sont implantées physiquement les données sur la machine.

Ces deux modes de passage des paramètres sont présents dans des langages comme C++, java, Ada, Visual-Basic .net, Delphi et C#. Il suffit donc pour le débutant, de bien comprendre le processus avec le pascal et par analogie il pourra l'utiliser avec les autres langages.

Exemple :

```
procedure B1 (x : integer; var y : integer) ;
begin
  y := 10*x
end ;

procedure B2 (x : integer; y :integer) ;
begin
 y := 10*x
end ;

procedure P ;
  var a , b: integer ;
begin
  a := 100 ; b := 0 ;
  B1 ( a , b ) ;

  a := 100 ; b := 0 ;
  B2 ( a , b ) ;

end ;
```

Dans la procédure B1
x est passé par valeur
y est passé par référence

Dans la procédure B2
x est passé par valeur
y est passé par valeur

Dans la procédure P

Appel de B1
B1(valeur a , ref b)
Résultat après appel :
b = 1000

Appel de B2
B2(valeur a , valeur b)
Résultat après appel :
b = 0

7. Fonction ou procédure ?

Une fonction est un bloc de programme qui réalise des traitements et renvoie une valeur unique, c'est une procédure ne possédant qu'un seul élément de sortie (appelé paramètre).

Tout ce qui a été énoncé sur les procédures **s'applique** in extenso aux **fonctions**.

La ligne : "**procedure B (x , y : integer ; var t : integer) ;** "
se dénomme l'en-tête de la procédure.

La ligne : "**function B (x , y : integer) : integer ;** "
se dénomme l'en-tête de la fonction

En pascal les blocs peuvent être implémentés aussi par des fonctions mais **uniquement** lorsqu'il n'y a qu'une donnée de sortie (un seul résultat).

Exemple1 :

```
function B1 (x : integer ) : integer ;
begin
  result := 10*x
end ;

procedure P ;
  var
     a , b: integer ;
begin
  a := 100 ;
  b := B1 ( a )
end ;
```

Dans la function B1
x est passé par valeur
B1 renvoie un résultat de type integer

Dans la procédure P

Appel de la fonction B1
b := B1(valeur a)
Résultat après appel :
b = 1000

Exemple2 :

```
function TTC (PHT,Tva : real ) : real ;
begin
  result := PHT*Tva
end ;


procedure CalculPrix ;
  var  PrixHT , PrixTTC : real ;
begin
 PrixHT:= 100 ;
 PrixTTC := TTC ( PrixHT , 1.186 )
end ;
```

Dans la function TTC
PHT et Tva sont passés par valeur
TTC renvoie un résultat de type real qui est
Le paramètre prix hors taxe multiplié
par le paramètre taux de TVA.

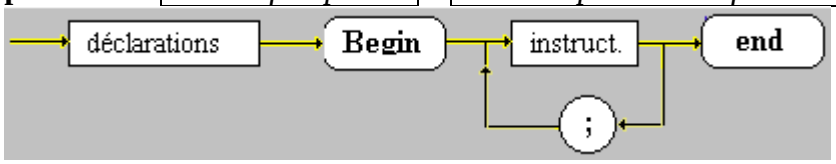
Dans la procédure CalculPrix

PrixHT = 100 €
Appel de la fonction TTC
PrixTTC := TTC (valeur a , 1.186)
Résultat après appel :
PrixTTC = 118,6 €

Les déclarations de fonctions et de procédures suivent le schéma grammatical de la déclaration générale du programme principal :

< *Déclaration de procédure* >

procedure *<identif de proc>* *<liste de paramètres formels>* **;**



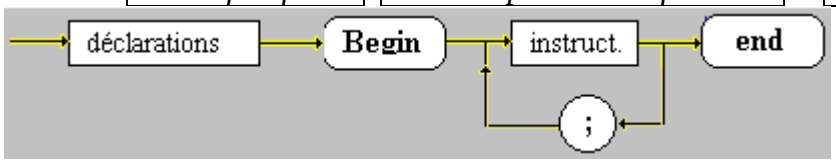
;

Exemple :

```
procedure    Calcul        (x : integer; var y :integer)    ;
             <identif de proc>   <liste de paramètres formels>

  var a, b : integer ; <déclarations>

  begin

    x := a*x  ; <instruction>
    y := x - b ; <instruction>

  end ;
```

< *Déclaration de fonction* >

function *<identif de fonc>* *<liste de paramètres formels>* **:** *<type du résultat>* **;**



;

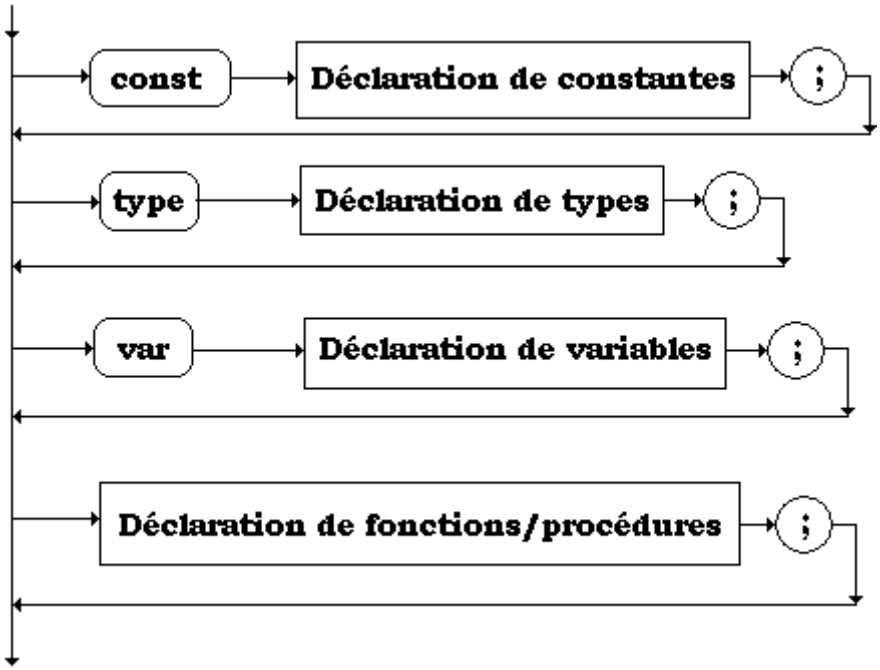
Exemple :

```
function     Calcul        (x : integer)     :     integer  ;
             <identif de fonc>  <liste de paramètres formels>  <type du résultat>

  var a : integer ; <déclarations>

  begin

    result := a*x  ; <instruction>

  end ;
```

8. Visibilité des variables

Le langage pascal suivant méthode de la programmation structurée descendante, les déclarations de fonctions/procédures peuvent être imbriquées :

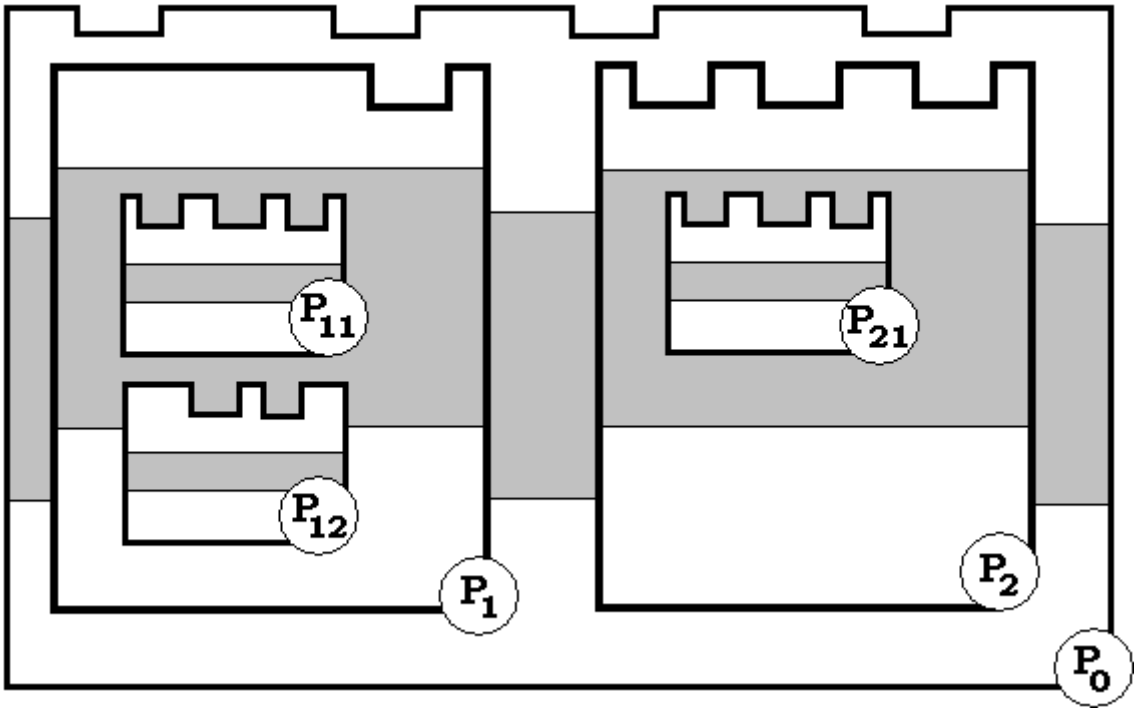
< *Déclarations* > :



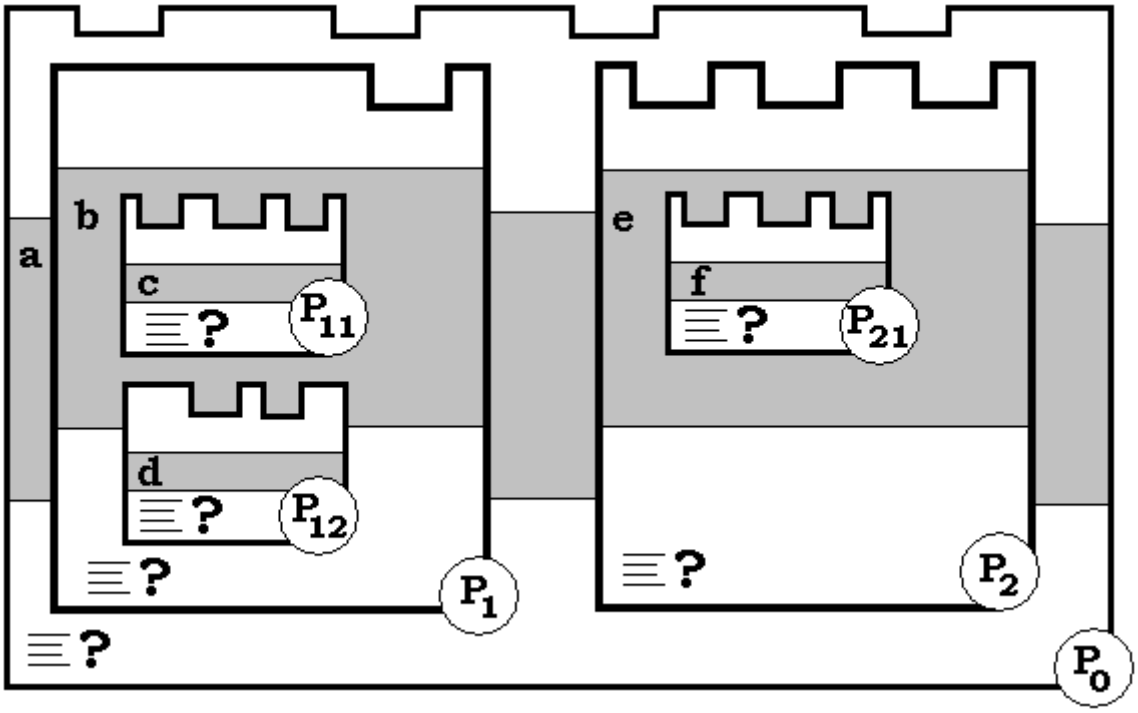
Exemple de déclarations imbriquées dans la même procédure P0 :

```
procedure P0 (x,y,z : char) ;
 var  a , b: integer ;

    procedure P1 ( var u : integer) ;
      var  a , b: integer ;
        procedure P11 ( var u,v,w : integer) ;
         var  a , b: integer ;
        begin
          …..
        end ;

        procedure P12 ( t : integer; h :char) ;
         var  a , b: integer ;
        begin
          …..
        end ;
    begin { P1 }
       …..
    end ; { P1 }

  procedure P2 ( f, g, h : real) ;
    var  a , b: integer ;
      procedure P21 ( n, m, p : integer) ;
        var  a , b: integer ;
      begin
        …..
      end ;

    begin  { P2 }
     …..
    end ; { P2 }

begin  { P0 }
  …..
end ; { P0 }
```

Ecritures que l'on peut représenter schématiquement par les imbrications de blocs qui suivent (les parties grisées d'un bloc correspondent à la partie déclaration du bloc) :



Supposons dans l'exemple précédent que la partie déclaration de chaque bloc contienne outre l'éventuelle déclaration d'un autre bloc, des déclarations de variables (**a** dans le bloc P0, **b** dans le bloc P1, **c** dans le bloc P11, **d** dans le bloc P12, **e** dans le bloc P2, **f** dans le bloc P21) :



Code pascal du schéma précédent :

```
procedure P0 (x,y,z : char) ;
 var  a : integer ;

    procedure P1 ( var s : integer) ;
      var  b : integer ;
        procedure P11 ( var u,v,w : integer) ;
         var  c : integer ;
        begin
          ….. ?….
        end ;

        procedure P12 ( t : integer; h :char) ;
         var  d : integer ;
        begin
          ….. ?….
        end ;
    begin  { P1 }
       ….. ?….
    end ; { P1 }

  procedure P2 ( f, g, h : real) ;
    var  e : integer ;
      procedure P21 ( n, m, p : integer) ;
        var  f : integer ;
      begin
        ….. ?….
      end ;

    begin  { P2 }
     ….. ?….
    end ; { P2 }

begin  { P0 }
  ….. ?….
end ; { P0 }
```

Etant donné les possibilités offertes par cette disposition des blocs en Pascal, il vient immédiatement une question sur les accès autorisés ou non aux données situées dans les parties déclarations des blocs P0, P1, etc...

En d'autres termes, dans la partie code de chaque bloc quelles variables peut-on utiliser ? Par exemple dans le corps (la partie code) de la procédure P12 peut-on utiliser toutes les variables **a, b, c, d, e, f** ou bien seulement certaines et selon quelles règles ?

```
procedure P12 ( t : integer; h :char) ;
 var  d : integer ;
 begin
   ….. ?….
 end ;
```

Ces autorisations d'accès aux données situées dans des blocs imbriquées sont contenues dans la notion de règle de visibilité dans les langages à structure de bloc (Pascal en est un cas particulier, ces règles s'appliqueront aussi à d'autres langages)

Règle de visibilité:

Toute donnée X déclarée localement dans un bloc P_k est n'est visible que :

- dans le bloc où elle est déclarée,
- et dans tous les blocs P_{k+n} imbriqués dans P_k.
- Un paramètre formel est considéré comme une variable locale au bloc.

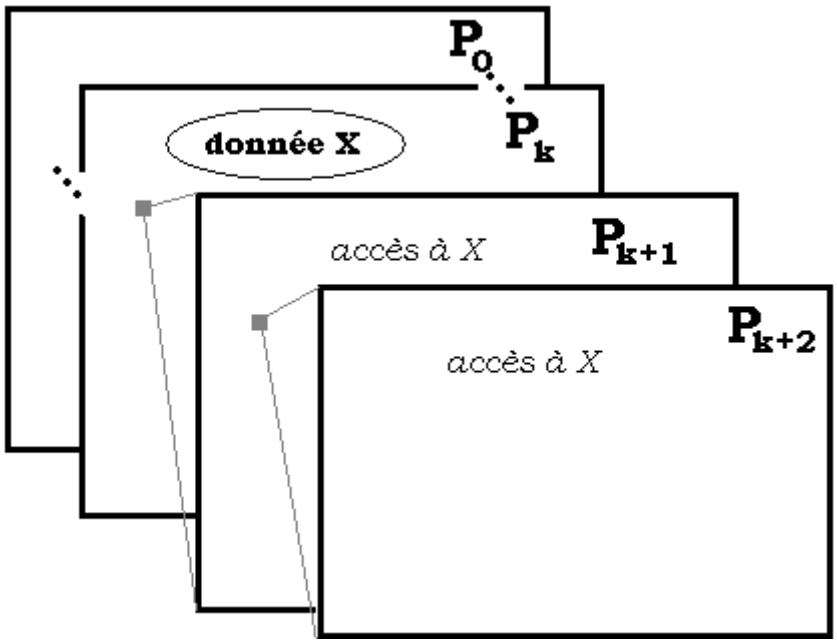


fig - visibilité d'une donnée X déclarée dans P_k

Remarque : masquage

Lorsqu'une donnée déclarée sous le nom X dans un bloc P_k est redéclarée sous le même nom X dans un bloc P_{k+n} imbriqué dans P_k, la donnée X de P_{k+n} masque les informations contenues dans la donnée X de P_k dans le bloc P_{k+n} et dans ceux qu'ils contient.

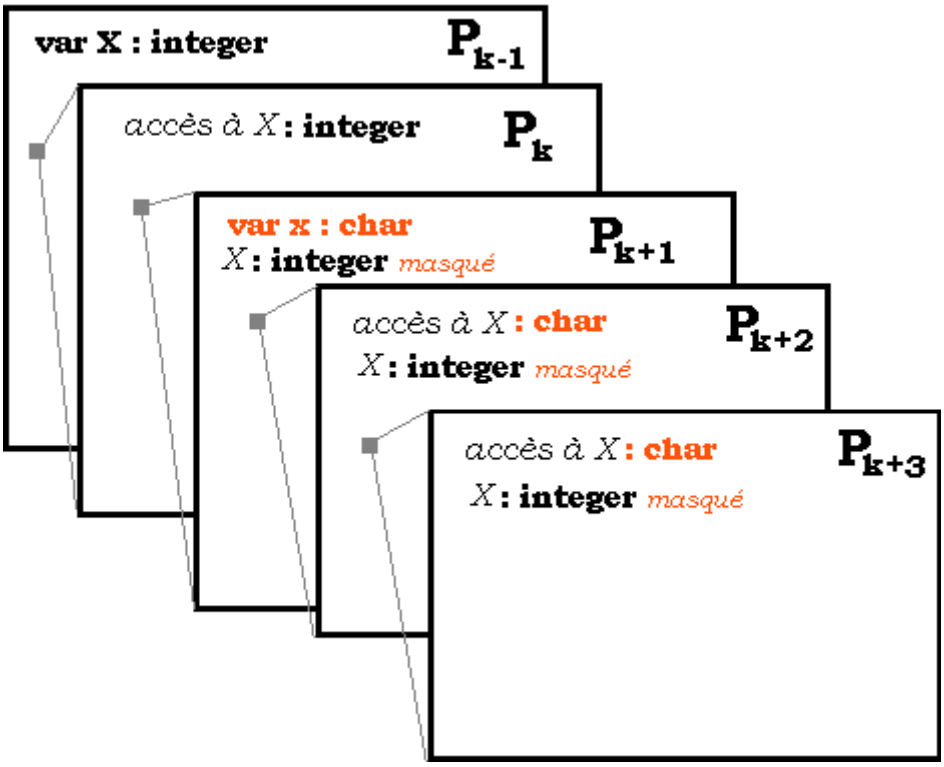


fig - visibilité d'une donnée X déclarée dans P_{k-1}

Etudions la visibilité des variables **a, b, c, d, e, f** dans les blocs P0, P1, P11, P12, P2, P21 figurées ci-dessous :

```
procedure P0 (……) ;
 var  a : integer ;

    procedure P1 ( …….) ;
      var  b : integer ;
        procedure P11 ( ……) ;
         var  c : integer ;
        begin
          ….. ?….
        end ;

        procedure P12 ( ……) ;
         var  d : integer ;
        begin
          ….. ?….
        end ;
    begin  { P1 }
       ….. ?….
    end ; { P1 }

  procedure P2 ( ……. ) ;
    var  e : integer ;
      procedure P21 ( …….. ) ;
        var  f : integer ;
      begin
        ….. ?….
      end ;

    begin  { P2 }
     ….. ?….
    end ; { P2 }

begin  { P0 }
  ….. ?….
end ; { P0 }
```

Etablissons à partir de la règle de visibilité énoncée plus haut, deux tableaux récapitulatifs croisés de la visibilité des variables **a, b, c, d, e, f** :

variable	Bloc où cette variable est visible
a	P0, P1, P11, P12, P2, P21
b	P1, P11, P12
c	P11
d	P12
e	P2, P21
f	P21

Bloc	variables visibles dans ce bloc
P0	**a**
P1	**a, b**
P11	**a, b, c**
P12	**a, b, d**
P2	**a, e**
P21	**a, b, f**

Nous pouvons donc répondre maintenant aisément à la question posée plus haut : quelles variables peut utiliser dans la procédure P12 ?

La procédure P12 accède aux variables **a, b** et **d**, (avec en plus comme variables locales ses paramètres formels **t** et **h**) :

```
procedure P12 ( t : integer; h :char) ;
  var  d : integer ;
 begin
   // accès aux variables a, b, d, t et h,
 end;
```

9. Variables dynamiques, références ou pointeurs

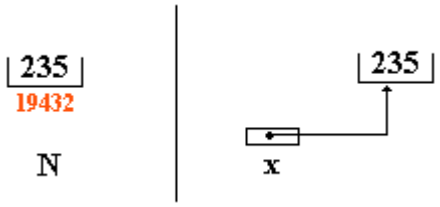
Définition

Beaucoup de langages disposent de la notion de pointeur C++ en particulier. C'est une notion proche de la machine qui a été utilisée dès le début pour représenter dans un programme l'allocation dynamique de mémoire. Dans une structure à allocation dynamique de mémoire le compilateur ne connaît pas à l'avance la taille de la structure, la gestion de la mémoire est alors confiée au programmeur. C'est lors de l'exécution et au fur et à mesure des mises à jours que la taille de la structure varie, comme par exemple dans la gestion d'une liste dont la taille varie en fonction des ajouts ou des suppressions. A l'opposé, une structure statique est une entité dont le compilateur connaît très exactement la taille avant l'exécution du programme, comme par exemple la structure de données de type tableau peut être considérée comme une structure statique puisque la taille du tableau (nombre de cellules) est connue lors de la déclaration.

En fait, les langages récents ne disposent plus de cette notion de pointeurs ou variables dynamiques parce qu'à l'usage elle s'est révélée dangereuse car trop proche de la machine laissant le programmeur se débrouiller seul avec la gestion de la mémoire, elle est utilement remplacée par la notion de référence d'objet comme dans Java, le langage C# demandant une autorisation pour traiter du code non sûr (**unsafe** code). Delphi quant à lui, combine les deux outils : pointeurs et références d'objet,la version Delphi 8.Net adoptant la même démarche que C# (**unsafe** code).

La notion de pointeur très présente, voir même essentielle dans un langage comme le C, est utilisable en pascal.

Prenons par exemple une variable numérique N entière d'adresse en mémoire centrale 19432 et contenant le nombre entier 235, nous appelons x un pointeur vers cette variable N, une variable dynamique contenant l'adresse de la variable N :



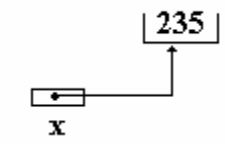
Nous dirons aussi que x « pointe » vers la variable N et que le « contenu » de x est 235.

En pascal (utiliser Delphi en mode console), une variable dynamique se déclare comme une variable classique mais le type est précédé du symbole « ^ », elle est typée (le type de la donnée vers laquelle elle pointe), mais sa gestion est entièrement à la charge du programmeur à travers les procédures d'allocation et de désallocation mémoire respectivement appelées **new** et **dispose**.

Utilisation pratique des variables dynamiques

Contenu d'une variable dynamique « x » déjà allouée : il est noté **« x^ »**

Dans l'exemple précédent :

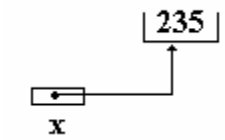


x^ vaut 235 (contenu de la variable dynamique)

x vaut 19432 (adresse de la variable dynamique)

Détaillons pas à pas un programme d'utilisation de pointeur

Soit l'exemple précédent :



Le programme de droite écrit sur l'écran le « contenu » de la variable x (contenu de la cellule pointée par x) soit : **x vaut : 235**.

Voici le programme à analyser :

```
program VarDyn;
 var
  x : ^integer;
  begin
   new(x);
   x^:= 235;
   writeln('x vaut: ',x^);
   dispose(x);
  end.
```

Déclaration d'une variable dynamique « x » de type entier :

Soit l'instruction :

var x : ^integer ;

Résultat produit :

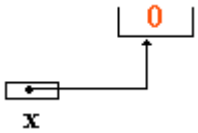
x est créée (mais x ne pointe vers rien encore)
x vaut nil

Allocation d'une variable dynamique « x » déjà déclarée :

Soit l'instruction :

new (x) ;

Résultat produit :



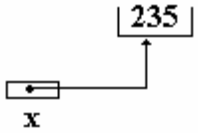
une cellule mémoire de type integer est crée,
x pointe vers la cellule créée.
(x vaut la valeur de l'adresse de la cellule)

Affectation du contenu d'une variable dynamique « x » déjà déclarée :

Soit l'instruction :

x^ := 235 ;

Résultat produit :



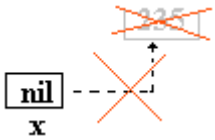
La cellule mémoire pointée par x contient 235.

Désallocation d'une variable dynamique « x » déjà allouée :

Soit l'instruction :

dispose (x) ;

Résultat produit :



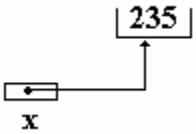
La cellule mémoire qui contenait 235 n'existe plus, elle est rendue au système (ont dit désallouée)

Attention

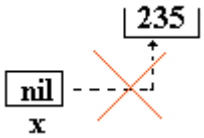
Ne pas confondre l'effacement de l'adresse d'une variable dynamique et sa désallocation.

Effacement de l'adresse d'une variable dynamique : mot clef **« nil »**
Désallocation d'une variable dynamique : procédure **dispose(…)**

Soit l'exemple précédent :

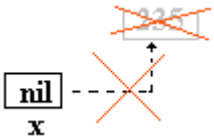


Résultat produit par x := nil **:**



- **x^** n'existe plus (x ne pointe vers plus rien)
- **x vaut nil**
- La cellule mémoire qui contient 235 **existe toujours**, mais n'est plus accessible !

Résultat produit par dispose (x) **:**



- **x^** n'existe plus (x ne pointe vers plus rien)
- **x vaut nil**
- La cellule mémoire qui contenait 235 **n'existe plus** !

C'est en particulier cette dernière remarque qui pose le plus de soucis de maintenance aux développeurs utilisant les pointeurs (par ex : problème de la référence folle).

Affectation de variables dynamiques entre elles :

On suppose que deux variables dynamiques « x et y » de type **^integer** ont été déclarées et créées par la procédure **new**, nous figurons ci-après l'incidence de l'affectation x := y sur ces variables :

Soient les instructions :

x^ := 235 ;

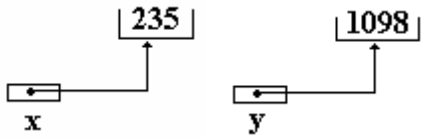
y^ := 1098 ;

Soient l'affectation :

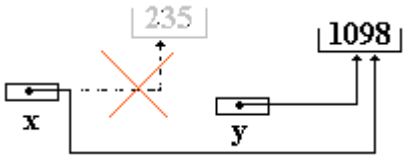
x := y ;

x et y pointent vers la même cellule mémoire

Résultat produit :

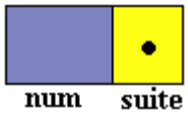


Résultat produit :

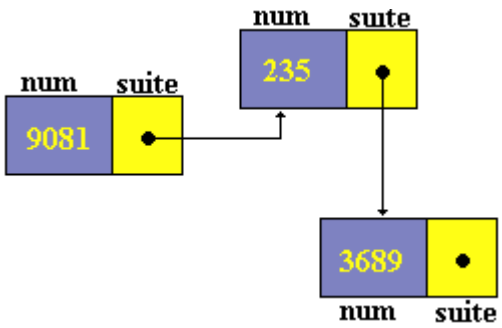


Une structure de données récursive avec pointeurs

Prenons une structure de données organisée sous forme de liste composée de cellules qui sont elles mêmes chacune un enregistrement (un **record**) contenant deux champs **num** et **suite** :



Le champ **num** est de type entier, et le champ **suite** est une variable dynamique de type cellule (lorsqu'il est alloué, il pointe donc vers une nouvelle cellule) :

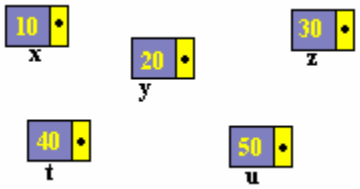


Soit un programme d'exemple de structure récursive (Delphi en mode console) utilisant les variables dynamiques pour représenter cette structure.

```
program pointeur;

 type

  cell = ^struct;

  struct = record
   num : integer;
   suite : cell
  end;

  var
   x , y, z , t , u : cell;
```

```
begin
   new(x) ;     x^.num := 10;
   new(y) ;     y^.num := 20;
   new(z) ;     z^.num := 30;
   new(t) ;     t^.num := 40;
   new(u) ;     u^.num := 50;
end.
```

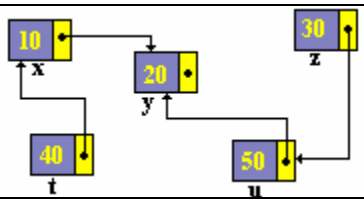
Ce programme crée 5 cellules :

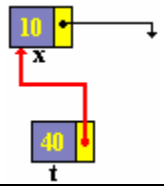
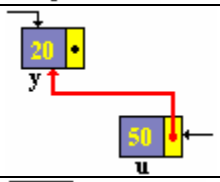
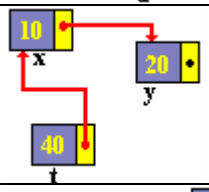
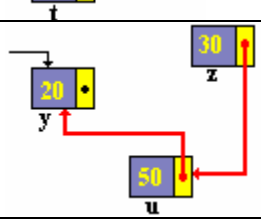


Les instructions suivantes :

```
t^.suite := x;
x^.suite := y;
z^.suite := u;
u^.suite := y;
```

représentent les liens ci-contre:



L'instruction suivante	Représente l'accès au lien	Et écrit sur la console
writeln (t ^. suite^. num);		10 (le contenu du champ num de x)
writeln (u^. suite^. num);		20 (le contenu du champ num de y)
writeln (t ^.suite^.suite^.num);		20 (le contenu du champ num de y)
writeln (z ^.suite^.suite^.num);		20 (le contenu du champ num de y)

La notion de référence est abordée au chapitre sur la programmation objet, c'est en fait un pointeur entièrement encapsulé sur lequel il n'est possible de faire qu'une seule opération : l'affectation de référence.

Exercices chapitre 2

Ex-1 : Au sujet des parenthèses bien formées dont on rappelle une C-grammaire :

G : $\mathbf{V_N}$ = {S}
$\mathbf{V_T}$ = { (,) }
Axiome : S
Règles 1 : S ⟶ (SS)S
2 : S ⟶ ε

1°) Proposez 3 autres C-grammaires engendrant le même langage de parenthèses.
2°) Construisez dans chacune d'elle l'arbre de dérivation du mot **((( )( )))(( )).**

Ex-2 : Soit G la C-grammaire suivante et L(G) le langage engendré par G :

G : V_N = { S, A ,B }
V_T = { **(** , **)** , **o** }
Axiome : A
Règles :
1 : A → **(** A
2 : A → **(** S
3 : S → **o** S
4 : S → **)** B
5 : B → **)** B
6 : B → **)**

1°) Donnez le mot le plus petit appartenant au langage L(G) (celui de longueur minimale).
2°) Donnez très précisément la forme générale des mots du langage L(G) (avec contraintes sur les indices lorsqu'il y en a).
3°) construisez l'arbre de dérivation dans G de la chaîne : $\mathbf{(}^{3}\ \mathbf{o}^{2}\ \mathbf{)}^{4}$

Ex-3 : Problème classique du **défaut de fermeture** en Algol, en Pascal en Java, en C# et autre... :

A - Soit G_0 une grammaire ambiguë de l'instruction if ...then...else en Pascal
$\mathbf{V_N}$ = {<Expr.> , S}
$\mathbf{V_T}$ = { **<u>if</u>** , **<u>then</u>** , **<u>else</u>** , **P** , **a** }
Axiome : S
Règles 1 : S ⟶ **<u>if</u>** < Expr.> **<u>then</u>** S
2 : S ⟶ **<u>if</u>** < Expr.> **<u>then</u>** S **<u>else</u>** S
3 : S ⟶ **a**
4 : < Expr.> ⟶ **P**

Donnez 2 arbres de dérivation dans G_0, de la chaîne :

<u>if</u> P **<u>then</u>** **<u>if</u>** P **<u>then</u>** a **<u>else</u>** a

B - On propose une autre grammaire ambiguë G_1 du même langage :
$\mathbf{V_N}$ = {<Expr.> , S , S'}
$\mathbf{V_T}$ = { **<u>if</u>** , **<u>then</u>** , **<u>else</u>** , **P** , **a** }
Axiome : S
Règles 1 : S ⟶ **<u>if</u>** < Expr.> **<u>then</u>** S S'
2 : S ⟶ **a**
3 : S' ⟶ **<u>else</u>** S
4 : S' ⟶ ε

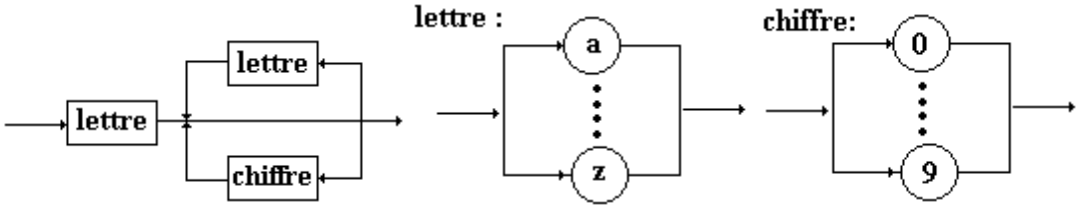
5 : < Expr.> ⟶ **P**

a) Refaire les 2 arbres de dérivation du mot « **if** P **then** **if** P **then** a **else** a » dans G_1.
b) Quelle amélioration est apportée par G_1 par rapport à G_0 ?

C - On construit une grammaire non ambiguë du langage :

a) Dans la grammaire G_0 , proposez une solution syntaxique permettant de lever l'ambiguïté en rajoutant un symbole supplémentaire dans $\mathbf{V_T}$ (grammaire G_2).
b) Montrer que le nouveau mot « **if** P **then** **if** P **then** .... » ne peut avoir qu'un seul arbre de dérivation dans G_2.

Ex-4 : ci dessous les diagrammes syntaxiques d'un identificateur dans un langage de programmation :



Ecrivez une grammaire en BNF traduisant ces diagrammes.

Ex-5 : Soit le programme Pascal suivant :

```
Program essai;
Const
  n = 50
Type
  tableau = array[1..n] of integer;
Var
  table : tableau ;

Procedure Lire( T : tableau );
begin
 for i:=1 to n do
  Readln( T[i] );
End;

begin
 Lire ( table);
 for i:=1 to n do
  write( T[i] ,'  ' );
end.
```

Que fait et qu'affiche très précisément ce programme ?

Ex-6 : Ecrire un programme Delphi console calculant la somme des 10 premiers termes de la série
$S_n = \Sigma\ 1/(2i+1)$, soit :S=1+1/3+1/5+1/7+...+1/19.Le programme affichera la somme S.

Ex-7 : Delphi possède une fonction LowerCase permettant de transformer tous les caractères d'une string en minuscule. Ecrivez votre propre fonction "Lowerstring (nom:string)" qui renvoie la string nom en minuscule sans utiliser la fonction LowerCase.

Ex-8 : Delphi possède les opérateurs booléens or, and et Xor. Ecrire un programme console affichant la table de vérité de chacun de ces trois opérateurs.

Ex-9 : Ecrivez les fonctions booléennes : "implique(p , q: boolean)" qui renvoie le résultat de p => q et "equivalent(p , q: boolean)" qui renvoie le résultat de p ⇔ q.

Réponses partielles:

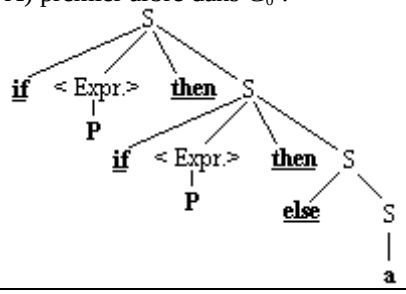
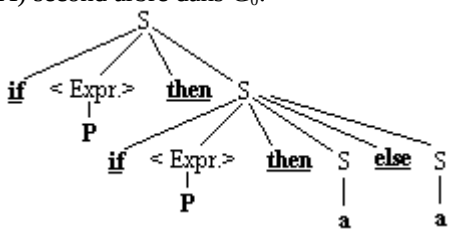
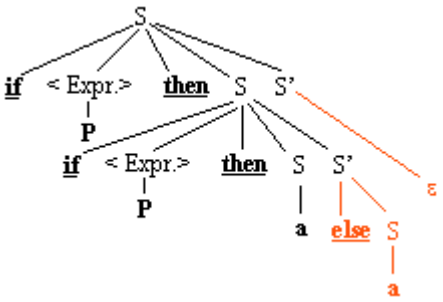
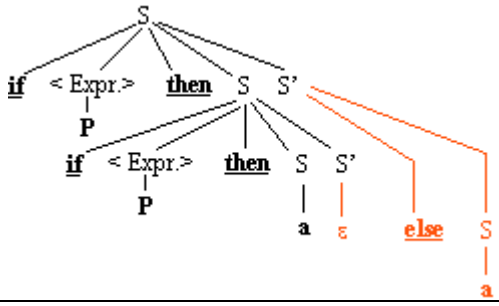
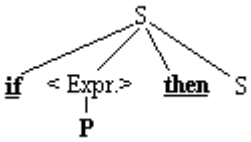
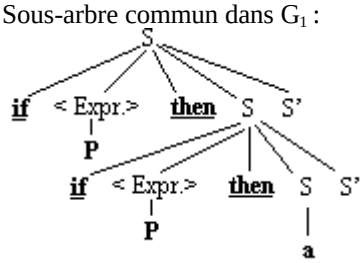
Ex-1 : **Règles**

1 : S ⟶ S(SS) 2 : S ⟶ ε	1 : S ⟶ (S)S 2 : S ⟶ ε	1 : S ⟶ S(S)S 2 : S ⟶ ε	*Etc..*

Ex-2 : 1°) mot minimal : **( ))**

2°) L(G) = { (^{n} o^p)^q , n ≥ 1, p ≥ 0, q ≥ 1 }

Ex-3 :

A) premier arbre dans G_0 : 	A) second arbre dans G_0: 
B) premier arbre dans G_1 : 	B) second arbre dans G_1 : 
Sous-arbre commun dans G_0 :  La grammaire G_0 permet une analyse moins profonde que G_1 avant que l'ambiguïté se révèle.	Sous-arbre commun dans G_1 : 

Soit G_2

V_N = {<Expr.> , S}

V_T = { **if** , **then** , **else** , **P** , **a** , **endif** }

Axiome : S

Règles 1 : S ⟶ **if** < Expr.> **then** S **endif**

2 : S ⟶ **if** < Expr.> **then** S **else** S **endif**

3 : S ⟶ **a**

4 : < Expr.> ⟶ **P**

On ne peut qu'écrire dans G_2 l'une ou l'autre des deux seules phrases distinctes suivantes :

- **if** P **then** **if** P **then** a **else** a **endif** **endif**
- **if** P **then** **if** P **then** a **endif** **else** a **endif**

Ex-4 : Soit une grammaire G_2 répondant à la question

V_N = {<identif.> , <lettre.> , <chiffre.> , <suite> } , **V_T** = { **a, b , … , z , 0 , … , 9** }

Axiome : <identif.>

Règles 1 : <identif.> ⟶ <lettre.> <suite>

2 : <suite> ⟶ <lettre.> <suite> | < chiffre.> <suite> | ε

3 : <lettre.> ⟶ **a | b | … | z**

4 : < chiffre.> ⟶ **0 | 1 | … | 9**

Ex-5 : Appel de la procédure Lire avec le paramètre table : Lire (table). La **Procedure** Lire(T : tableau) reçoit un paramètre passé par valeur, donc elle travaille sur une copie du tableau table en saisissant au clavier les données, mais lors de la fin de l'appel le tableau local est détruit et l'original n'a pas été modifié donc le tableau table est resté vide !

Ex-6 : Somme 1+1/3+1/5+1/7+...+1/19

Boucle for croissante	*Boucle for décroissante*

program for_do;
const max=20;
var
som : real;
i : integer;
begin
som:= 0;
for i := 0 **to** 9 **do**
som := som + 1 / (2*i+1);
writeln('somme = ', som);
end.
	program for_do;
	const max=20;
	var
	som : real;
	i : integer;
	begin
	som:= 0;
	for i := 9 **downto** 0 **do**
	som := som + 1 / (2*i+1);
	writeln('somme = ', som);
	end.

Ex-7 : chaine → en minuscule

```
function Lowerstring (ch : string) : string;
var
  i: integer;
  sortie: string;
begin
  sortie := '';
  for i := 1 to length(ch) do
    if ch[i] in ['A'..'Z'] then
          sortie := concat (sortie, chr(ord(ch[i]) + ord('a') - ord('A')))
    else
          sortie := concat(sortie, ch[i] );
    result := sortie
end;
```

Ex-8 : Tables de vérités

```
program TableVerite;
var
  a, b, c:boolean;
begin
 writeln('    table du Et :');
 writeln('    a      b      Et');
 writeln('-------------------------');
 for a := false to true do
   for b := false to true do
       writeln( a:7, b:7, a And b:7);
 writeln('*************************');
 writeln('    table du Ou :');
 writeln('    a      b      Ou');
 writeln('-------------------------');
 for a := false to true do
   for b := false to true do
       writeln( a:7, b:7, a or b:7);
 writeln('*************************');
 writeln('    table du Xor :');
 writeln('    a      b      Xor');
 writeln('-------------------------');
 for a := false to true do
   for b := false to true do
       writeln( a:7, b:7, a Xor b:7);
end.
```

Ex-9 : implication et équivalence

P => Q = non P ou Q	**P ⇔ Q = (P => Q) et (Q => P)**
function implique (p , q : boolean) : boolean; **begin** result := **not** p **or** q **end**;	**function** equivalent (p , q : boolean) : boolean; **begin** result := implique (p,q)**and** implique(q,p) **end**;

Chapitre 3 : Développer du logiciel avec méthode

3.1 Développement méthodique du logiciel

- la production du logiciel
- conception structurée descendante et machines abstraites
- notion d'algorithme
- un langage de description d'algorithmes le LDFA
- le dossier de programmation
- trace formelle d'un algorithme
- traducteur LDFA - Pascal
- facteurs de qualité du logiciel

Machines abstraites **: exemple de traitement sur les chaînes**

- cas où la version du pascal contient un type chaîne
- cas où la version du pascal ne contient pas de type chaîne
- programme pascal obtenu
- autres versions d'implantation en pascal

3.2.Modularité

- définition : B.Meyer
- la modularité en pascal avec les **Unit**

3.3. Complexité, tri, recherche

- Notions de complexité temporelle et spatiale
- Mesure de la complexité temporelle d'un algorithme
- Notation de Landau O(n)
-

Trier des tableaux en mémoire centrale

- Le Tri à bulles
- Le Tri par sélection

- Le ri par insertion
- Le Tri rapide QuickSort
- Le Tri par tas HeapSort

Rechercher dans un tableau

- Dans un tableau non trié
- Dans un tableau trié

Exercices: algorithmes et leur traduction

3.1 : développement méthodique du Logiciel

Plan du chapitre:

1.Historique des langages

Introduction

1. Production du logiciel

1.1 Génie logiciel
1.2 Cycle de vie ***du logiciel***
1.3 Maintenance *d'un logiciel*
1.4 Production industrielle *du logiciel*

2. Conception structurée descendante

2.1 Critère simple d'automatisation
2.2 Analyse méthodique descendante
2.3 Analyse ascendante
2.4 *Programmation descendante* avec retour sur un niveau
2.5 Machines abstraites *et niveaux logiques*

3. Notion d'ALGORITHME

3.1 Langage algorithmique
3.2 Objets de base *d'un langage algorithmique*
3.3 Opérations sur les objets de base *d'un langage algorithmique*

4. Un langage de description d'algorithme : LDFA

4.1 Atomes *du LDFA*
4.2 Information *en LDFA*
4.3 Vocabulaire terminal *du LDFA*
4.4 Instructions simples *du LDFA*

5. Le Dossier de développement

5.1 Enoncé et spécification
5.2 Méthodologie
5.3 Environnement
5.4 Algorithme en LDFA
5.5 Programme en langage Pascal

6. Trace formelle d'un algorithme

6.1 Espace d'exécution *d'une instruction composée*
6.2 Exemple *avec trace formelle*

7. Traducteur élémentaire LDFA - Pascal

7.1 Traducteur
7.2 Exemple
7.3 Sécurité et ergonomie

8. Facteurs de qualité du logiciel

Introduction

Le bon sens est la chose du monde la mieux partagée...la diversité de nos opinions ne vient pas de ce que les uns sont plus raisonnables que les autres, mais seulement de ce que nous conduisons nos pensées par diverses voies, et ne considérons pas les mêmes choses. Car ce n'est pas assez d'avoir l'esprit bon, mais le principal est de l'appliquer bien.
R Descartes Discours de la méthode, première partie, 1637.

Le développement méthodique d'un logiciel passe actuellement par une démarche de " descente concrète " de la connaissance que l'humain a sur la problématique du sujet, vers l'action élémentaire exécutée par un ordinateur. Le travail du programmeur étant alors ramené à une traduction permanente des actions humaines en actions machines (décrites avec des outils différents).

Nous pouvons en première approximation différentier cette " descente concrète " en un classement selon quatre niveaux d'abstraction :

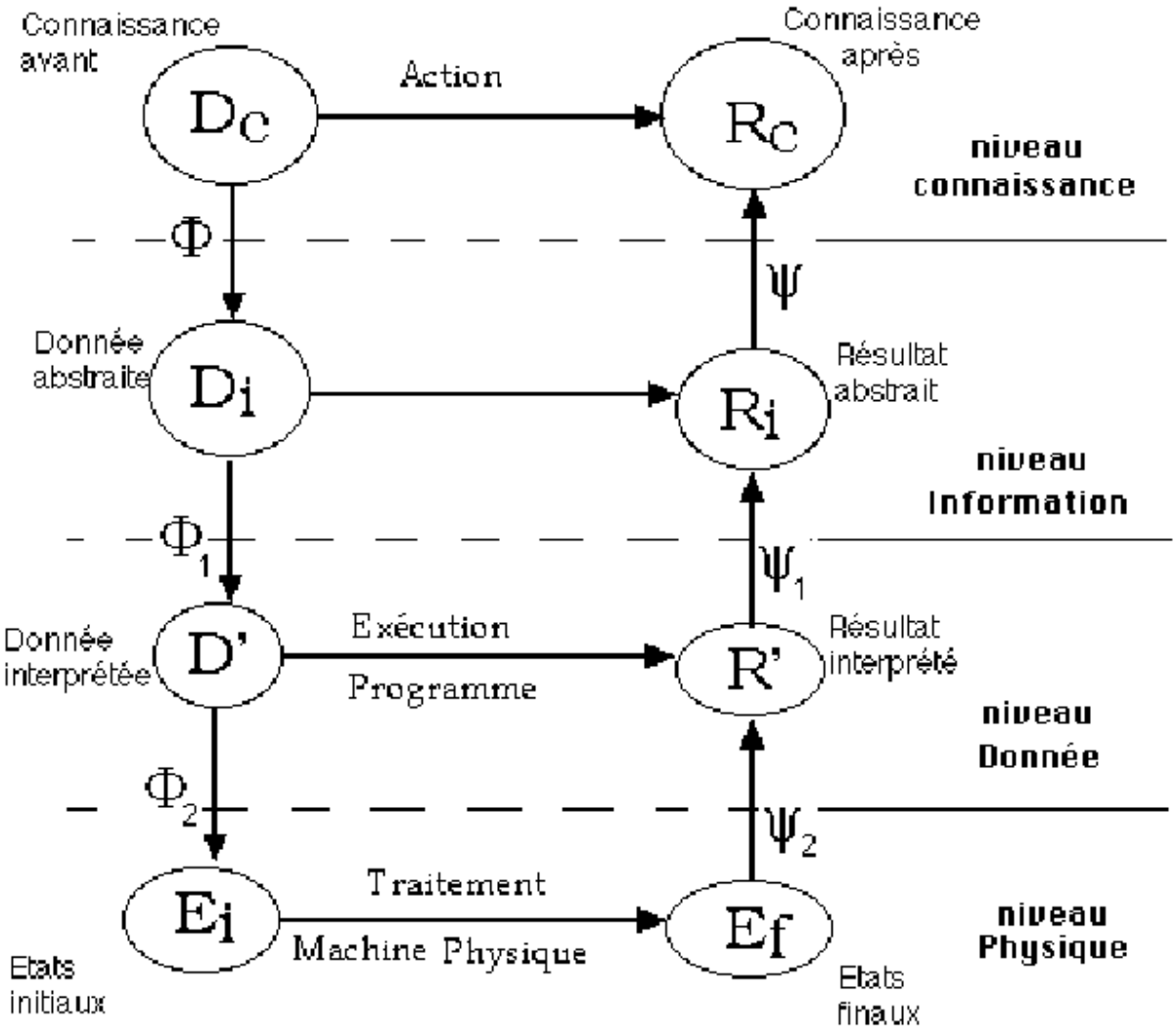


fig - schéma de descente concrète

Nous voyons que toute activité de programmation consiste à transformer un problème selon une descente graduelle de l'humain vers la machine. Ici nous avons résumé cette décomposition en 4 niveaux. La notion de programmation structurée est une réponse à ce type de décomposition graduelle d'un problème. L'algorithmique est la façon de décrire cette méthode de travail.

1. Production du logiciel

1.1 Génie logiciel

A une certaine époque, à ses débuts, l'activité d'écriture du logiciel ne reposait que sur l'efficacité personnelle du programmeur laissé pratiquement seul devant la programmation d'un problème.

De nos jours, le programmeur dispose d'outils et de méthodes lui permettant de concevoir et d'écrire des logiciels. Le terme **logiciel**, ne désigne pas seulement les programmes associés à telle application ou tel produit : il désigne en plus la documentation nécessaire à l'installation, à l'utilisation, au développement et à la maintenance de ce logiciel. Pour de gros systèmes, le temps de réalisation peut être aussi long que le temps du développement des programmes eux-mêmes.

Le **génie logiciel** concerne l'ensemble des méthodes et règles relatives à la production rationnelle des logiciels.

L'activité de développement du logiciel, vu les coûts qu'elle implique, est devenue une *activité économique* et doit donc être planifiée et soumise à des normes sinon à des attitudes équivalentes à celles que l'on a dans l'industrie pour n'importe quel produit.

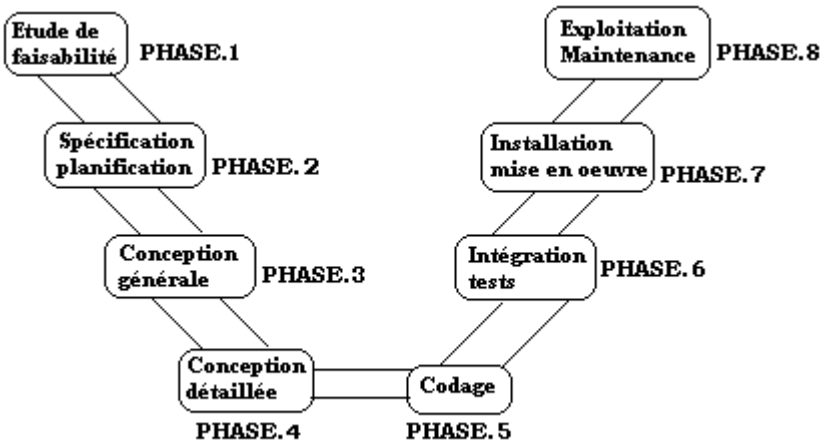
C'est pourquoi dans ce cours, le mot-clef est le mot **"composant logiciel"** qui tient à la fois de l'activité créatrice de l'humain et du composant industriel incluant une activité *disciplinée et ordonnée* basée pour certaines tâches sur des outils formalisés.

D'autre part le génie logiciel intervient lorsque le logiciel est trop grand pour que son développement puisse être confié à un seul individu ; ce qui n'est pas le cas pour des débutants, à qui il n'est pas confié l'élaboration de gros logiciels. Toutefois, il est possible de sensibiliser le lecteur débutant à l'habitude d'élaborer un logiciel d'une manière systématique et rationnelle à l'aide d'outils simples.

1.2 Cycle de vie du logiciel

Comme il faut un temps très important pour développer un grand système logiciel, et que d'autre part ce logiciel est prévu pour être utilisé pendant longtemps, on sépare fictivement des étapes distinctes dans ces périodes de développement et d'utilisation.

Le modèle dit de la cascade de Royce (1970) accepté par tout le monde informatique est un bon outil pour le débutant. S'il est utilisé pour de gros projets industriels, en supprimant les recettes et les validations en fin de chaque phase, nous disposons en initiation d'un cadre méthodologique. Il se présente alors sous forme de 8 diagrammes ou phases :



1.3 Maintenance d'un logiciel

Dans beaucoup de cas le coût du logiciel correspond à la majeure partie du coût total d'une application informatique. Dans ce coût du logiciel, la maintenance a elle-même une part prépondérante puisqu'elle est estimée de nos jours au minimum à **75% du coût total du logiciel**.

La maintenance est de trois sortes :

- adaptative (s'adapter à un nouvel environnement...)
- corrective (corrections d'erreurs...)
- perfective (améliorations demandées par le client...)

1.4 Production industrielle du logiciel

La production du logiciel étant devenue une activité industrielle et donc économique, elle n'échappe pas aux données économiques classiques. On répertorie un ensemble de caractéristiques associées à un projet de développement, chaque caractéristique se voyant attribuer un ratio de productivité.

Le ratio de productivité d'une caractéristique
C'est le rapport entre la productivité (exprimée en nombre d'Instructions Sources Livrées, par homme et par mois) d'un projet exploitant au mieux cette caractéristique, et la productivité d'un projet n'exploitant pas du tout cette caractéristique.

Le tableau suivant est tiré d'une étude de B.Boehm (Revue TSI 1982 : les facteurs de coût du logiciel):

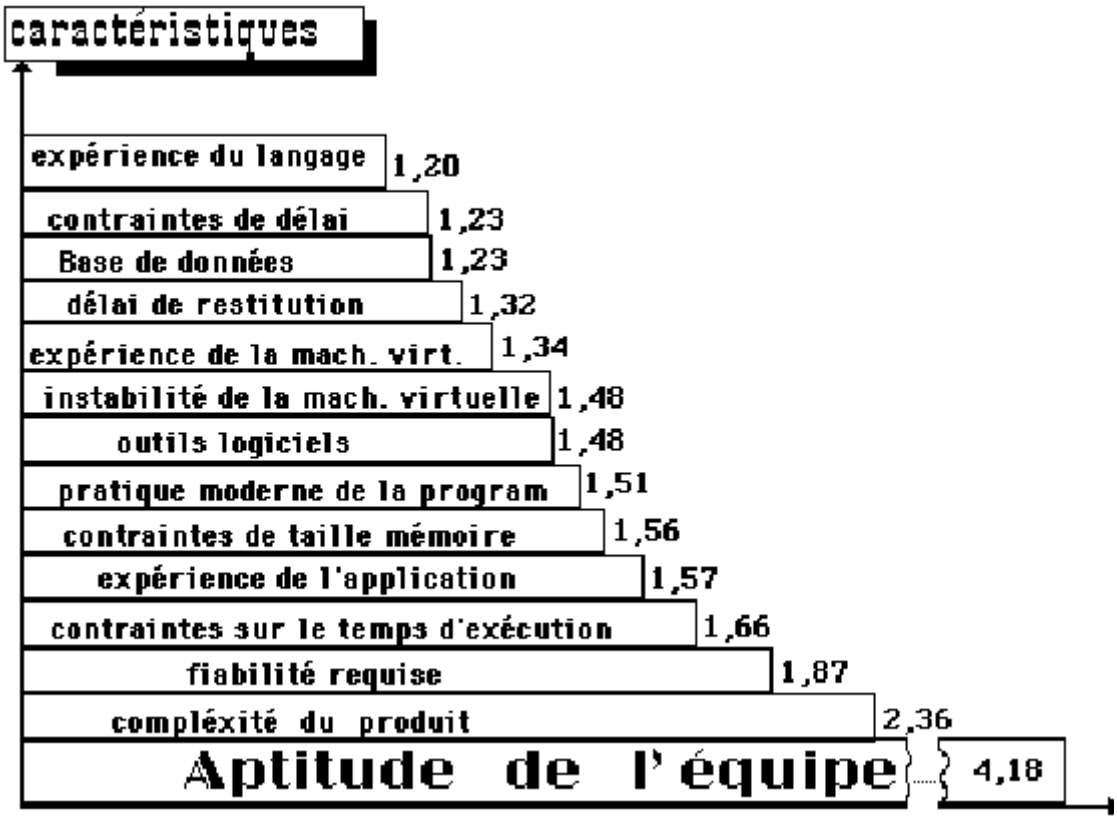


Tableau comparatif des divers ratios de productivité (B.Boehm)

Vous aurez remarqué en observant le graphique précédent que le facteur le plus important n'est pas l'expérience d'un langage (erreur commise par les néophytes). Ce qui explique entre autres arguments que l'enseignement de la programmation ne soit pas l'enseignement d'un langage.

Il apparaît que le facteur le plus coûteux reste un facteur sur lequel la technologie n'a aucune prise : l'aptitude qu'ont des individus à communiquer entre eux !

Pour l'élaboration d'un logiciel, nous allons utiliser deux démarches classiques : la méthode structurée ou algorithmique et plus tard une extension orientée objet de cette démarche.

2. Conception structurée descendante

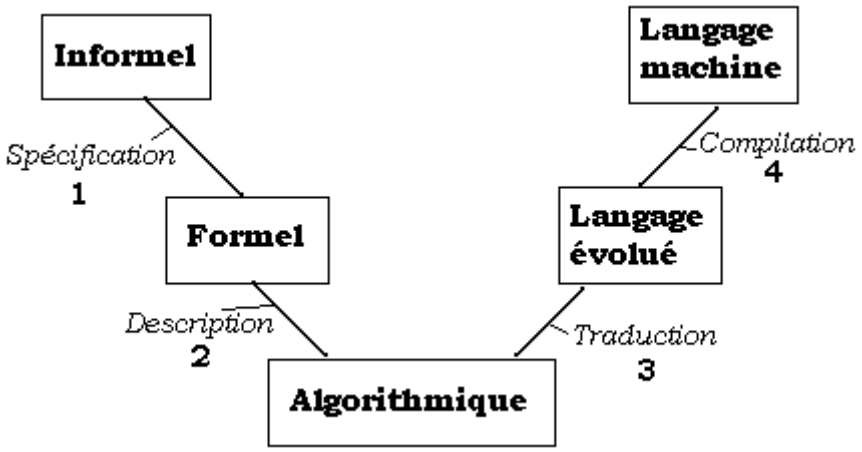
2.1 Critère simple d'automatisation

Un problème est **automatisable** (traitable par informatique) si :

- l'on peut parfaitement définir les données et les résultats,
- l'on peut décomposer le passage de ces données vers ces résultats en une suite d'opérations élémentaires dont chacune peut être exécutée par une machine.

Dans le cadre d'une initiation à la programmation, dans le cycle de vie déjà présenté plus haut, nous ne considérerons que les phases 2 à 6, en supposant que la faisabilité est acquise, et qu'enfin les phases de mise en œuvre et de maintenance sont mises à part.

Dans cette perspective, le schéma de la programmation d'un problème se réduit à 4 phases :



- La phase 1 de spécification utilisera les types abstraits de données (TAD),
- la phase 2 (correspondant aux phases 3 et 4 du cycle de vie) utilisera la méthode de programmation algorithmique,
- la phase 3 (correspondant à la phases 5 du cycle de vie) utilisera un traducteur manuel pascal,
- la phase 4 (correspondant à la phases 6 du cycle de vie) correspondra au passage sur la machine avec vérification et jeux de tests.

Nous utiliserons un " langage algorithmique " pour la description d'un algorithme résolvant un problème. Il s'agit d'un outil textuel permettant de passer de la conception humaine à la conception machine d'une manière souple pour le programmeur.
Nous pouvons résumer dans le tableau ci-dessous les étapes de travail et les outils conceptuels à utiliser lors d'une telle démarche.

ETAPES PRATIQUES	Matériel et moyens techniques à disposition
Analyse	Papier, Crayon, Intelligence, Habitude.
Mise en forme de l'algorithme	C'est l'aboutissement de l'analyse, esprit logique et rationnel.
Description	Utilisation pratique des outils d'une méthode de programmation, ici la programmation structurée.
Traduction	Transfert des écritures algorithmiques en langage de programmation.
Tests et mise au point	Mise au point du programme sur des valeurs tests ou à partir de programmes spécialisés.
Exécution	Phase finale : le programme s'exécute sans erreur.

2.2 Analyse méthodique descendante

Le second [précept], de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre.
R Descartes Discours de la méthode, seconde partie, 1637.

Définir le problème à résoudre:
expliciter les données
  préciser: leur nature
  leur domaine de variation
  leurs propriétés
expliciter les résultats
  préciser: leur structure
  leur relations avec les données
fin définir;
Décomposer le problème en sous-problèmes**;**
Pour chaque sous-problèmes identifié faire
  si solution évidente **alors** écrire le morceau de programme
  sinon appliquer la méthode au sous-problème
  fsi
fpour.

démarche proposée par J.Arsac

Cette démarche méthodique a l'avantage de permettre d'isoler les erreurs lorsqu'on en commet, et elles devraient être plus rares qu'en programmation empirique (anciens organigrammes).

Il apparaît donc plusieurs niveaux de décomposition du problème (niveaux d'abstraction descendants). Ces niveaux permettent d'avoir une description de plus en plus détaillée du problème et donc de se rapprocher par raffinements successifs d'une description prête à la traduction en instructions de l'ordinateur.

Afin de pouvoir décrire la décomposition d'un problème à chaque niveau, nous avons utilisé un langage algorithmique (et non pas un langage de programmation) qui emprunte beaucoup au langage naturel (le français pour nous).

2.3 Analyse ascendante

Le troisième [précept], de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu, comme par degrés, jusqu'à la connaissance des plus composés; et supposant même de l'ordre entre ceux qui ne se précèdent point naturellement les uns les autres.
R Descartes Discours de la méthode, seconde partie, 1637.

Nous essaierons de partir de l’existant (les fichiers sources déjà écrits sur le même sujet) et de reconstruire par étapes la solution. Le problème dans cette méthode est d’assurer une bonne cohérence lorsque l’on rassemble les morceaux.

Les méthodes objets que nous aborderons plus loin, sont un bon exemple de cette démarche. Nous n'en dirons pas plus dans ce paragraphe en renvoyant le lecteur intéressé au chapitre de la programmation orientée objet de cours.

2.4 Programmation descendante avec retour sur un niveau

Comme partout ailleurs, une attitude appuyée sur les deux démarches est le gage d'une certaine souplesse dans le travail. Nous adopterons une démarche d'analyse essentiellement descendante, avec la possibilité de remonter en arrière dès que le développement paraît trop complexe.

Nous adopterons dans tout le reste du chapitre une telle méthode descendante (avec quelques retours ascendants). Nous la dénommerons " programmation algorithmique ".

Nous utilisons les concepts de **B.Meyer** pour décomposer un problème en niveaux logiques puis en raffinant successivement les différentes étapes.

2.5 Machines abstraites et niveaux logiques

Principe :
On décompose chacune des étapes du travail en niveaux d'abstractions logiques. On suppose en outre qu'à chaque niveau logique fixé, il existe une machine abstraite virtuelle capable de comprendre et d'exécuter la description du problème sous la forme algorithmique en cours. Ainsi, en descendant de l'abstraction vers le concret, on passe graduellement d'un énoncé de problème au niveau humain à un énoncé du même problème à un niveau où la machine devient capable de l'exécuter.

Niveau logique	Machine abstraite	Enoncé du problème en
0	M_0 = l'humain	A_0 = langage naturel
1	M_1 = mach. Abstraite	A_1 = lang.algorithmique
. . .	. . .	. . .
n	M_n = machine+OS	A_n = langage évolué
n+1	M_{n+1}= machine physique	A_{n+1}= langage binaire

A partir de cette décomposition on construit un " arbre " de programmation représentant graphiquement les hiérarchies des machines abstraites.

Voici un exemple d'utilisation de cette démarche dans le cas de la résolution générale de l'équation du second degré dans **R**.

Le problème se décompose en deux sous-problèmes " :

- le premier concerne la résolution d'une équation du premier degré strict
- le second est relatif à la " résolution d'une équation du second degré strict " .

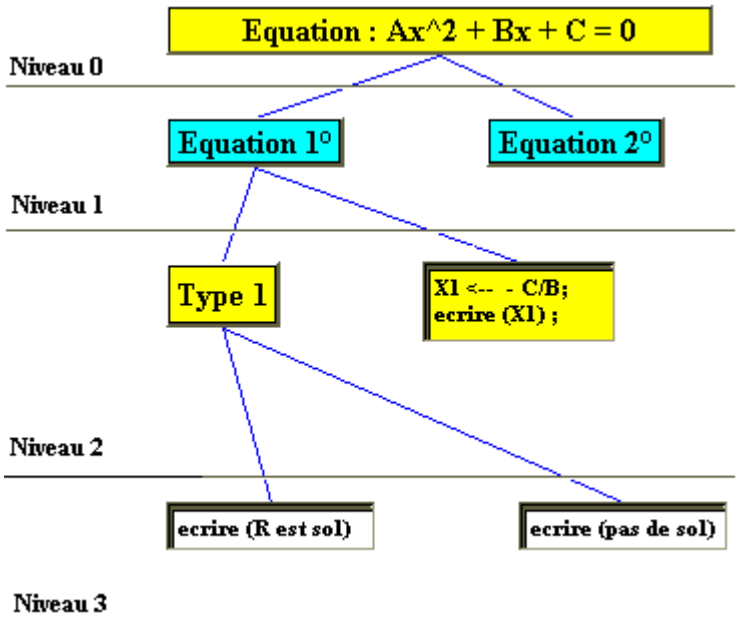


figure de la branche d'arbre 1er degré

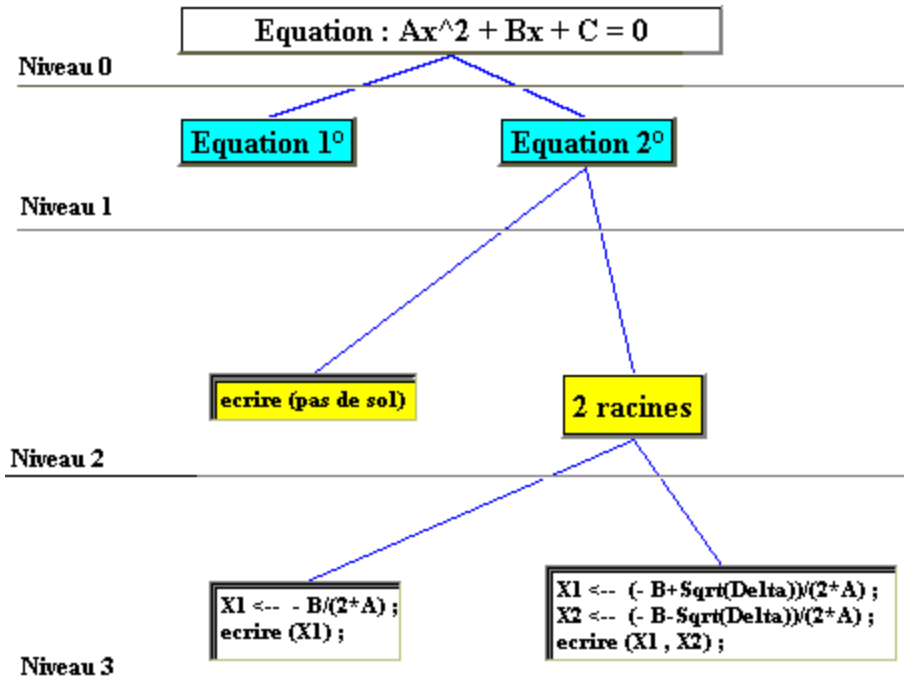


figure de la branche d'arbre 2ème degré

Nous avons utilisé comme langage de description des étapes intermédiaires un langage algorithmique basé sur des mots du français. Nous le détaillerons plus tard.

3. Notion d'ALGORITHME

Définition *(D.E. Knuth)*

Un algorithme est un ensemble de règles qui décrivent une séquence d'opérations en vue de résoudre un problème donné bien spécifié. Un algorithme doit répondre aux 5 caractéristiques suivantes :

- La finitude
- La précision
- Le domaine des entrées
- Le domaine des sorties
- L'exécutabilité

Notons qu'un algorithme exprime donc un procédé séquentiel (or dans la vie courante tout n'est pas nécessairement séquentiel comme par exemple écouter un enseignement et penser aux prochaines vacances), et ne travaille que sur des problèmes déjà transformés de la phase 1 à la phase 2 (la spécification). I

Il n'est pas demandé aux débutants de travailler sur cette étape du processus. C'est pourquoi la plupart des exercices de débutant sont déjà spécifiés dans l'énoncé, ou bien leur spécification est triviale.

Indiquons les éléments de définition des cinq autres caractéristiques demandées à un algorithme :

- **Finitude** : Le nombre d'étapes d'un algorithme doit être fini. Le temps d'exécution pourra être évalué.
- **Précision** : Chaque étape doit être parfaitement définie. Toutes les actions élémentaires doivent être connues.
- **Domaine des entrées** : Le champ des données d'entrée doit être spécifié.
- **Domaine des sorties** : Un algorithme ayant un résultat, il faut donner les champs correspondants aux résultats de sortie, ou du moins les relations entre les données d'entrée et les données de sortie.
- **Exécutabilité** : Un algorithme doit déboucher sur un programme exécutable en un temps fini et raisonnable.

Environnement

On appelle environnement d'un algorithme l'ensemble des entités utilisés par le processeur pendant le déroulement de l'algorithme.

Nous allons définir un langage de description des algorithmes qui nous permettra de décrire les arbres de programmation et le fonctionnement des machines abstraites de la programmation structurée.

Voici classiquement ce que tous les auteurs utilisent comme système de description d'un algorithme lorsqu'ils le font avec un langage. *Les deux sous-paragraphes qui suivent, fournissent les définitions des éléments fondamentaux d'un tel langage algorithmique, le paragraphe d'après construit un langage algorithmique fondé sur ces éléments fondamentaux*.

Nous verrons que l'algorithmique est par nature plus proche de l'étudiant que la machine. En effet dans la suite du cours, l'étudiant s'apercevra par exemple, que les nombres rationnels ne sont pas représentables simplement en machine, encore moins les nombres réels. Les langages d'implémentations impératifs comme Pascal, Java, C# etc… étant relativement pauvres à cet égard.

L'étudiant ne doit pas croire que l'informatique s'est résignée à ne travailler que sur les entiers et les décimaux, mais plutôt se rendre compte qu'il existe une palette importante de certains produits informatiques qui traitent plus ou moins efficacement les insuffisances des langages classiques par exemple vis à vis des rationnels (les systèmes de calcul formel comme MAPLE (étudié en Taupe),MATHEMATICA,... sont une réponse à ce genre d'insuffisance). Nous ne nous préoccupons absolument pas, dans un premier temps en algorithmique, ni de la vérification, ni du contrôle, ni des restrictions d'implantation des données. Notre préoccupation première est d'écrire des algorithmes justes qui fonctionnent sur des données justes.

3.1 Objets de base d'un langage algorithmique

Contenant

Nous appelons **contenant** toute cellule mémoire d'une machine abstraite d'un niveau fixé.

Contenu

Nous appelons **contenu** l'information représentée par l'état du contenant.

Atomes

Pour un contenant fixé on note A l'ensemble de tous ses états possibles, on dit aussi ensemble des **atomes** du niveau n (niveau du contenant).

Remarques :

a) un atome de niveau n est donc un état possible d'un contenant,
b) pour un niveau logique fixé, il y a un nombre d'atomes fini,
c) lorsque l'on est au niveau machine :

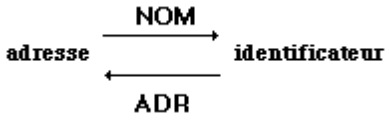
- le contenant est à p positions binaires(p est le nombre de bits du mot, p>1).
- A ={0,1}x....x {0,1} , p fois

Adresse fictive

Toute machine abstraite de niveau fixé dispose d'autant de cellules mémoires que nécessaire. Elles sont repérées par une **adresse fictive** (à laquelle nous n'avons pas accès).

Nom

Par définition, à toute adresse nous faisons correspondre bijectivement par l'opération **nom**, un identificateur unique définissant pour l'utilisateur la cellule mémoire repérée par cette adresse :



Nous définissons aussi un certain nombre de fonctions :

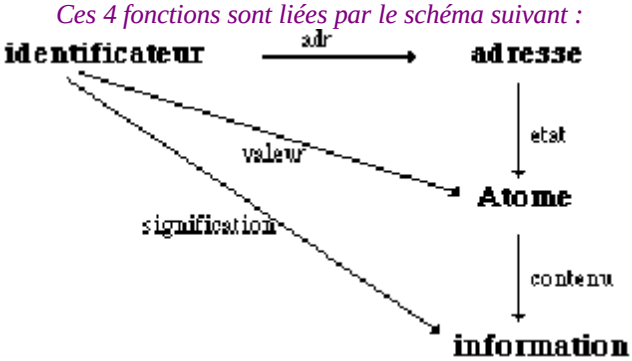
Etat : Adresse →Atome *(donne l'état associé à une adresse)*

valeur: identificateur →Atome *(donne l'état associé à un identificateur, on dit la valeur)*

contenu: Atome →information *(donne le contenu informationnel de l'atome)*

signification: identificateur →information *(sémantique de l'identificateur)*

Ces 4 fonctions sont liées par le schéma suivant :



3.2 Opérations sur les objets de base d'un langage algorithmique

Les parenthèses d'énoncé en LDFA seront algol-like : nous disposerons d'un marqueur du genre **debut** et d'un second du genre **fin** .

Exécutant ou processeur algorithmique

Nous appelons **exécutant** ou **processeur**, la partie de la machine abstraite capable de lire, réaliser, exécuter des opérations sur les atomes de cette machine, ceci à travers un langage approprié.

***Remarque**: l'opérateur formel **exécutant** dépend du temps.*

Instruction simple

C'est une instruction exécutable en un temps fini par un processeur et elle n'est pas décomposable en sous-tâches exécutables ou en autres instructions simples. Ceci est valable à *un niveau fixé*.

Instruction composée

C'est une instruction simple, ou bien elle est décomposable en une suite d'instructions entre parenthèses.

Composition séquentielle

Si i,j,...,t représentent des instructions simples ou composées, nous écrirons la composition séquentielle avec des " ; ". La suite d'instructions " i ; j; ..... ; t " est appelée une **suite d'instructions séquentielles**.

Schéma fonctionnel

C'est :

- soit un identificateur,
- soit un atome,
- soit une application **f** à n variables (où n>0):
  f : $(\text{identificateur})^n \rightarrow$ identificateur

Espace d'exécution

L'espace d'exécution d'une instruction, c'est le n-uplet des n identificateurs ayant au moins une occurrence dans l'instruction (ceci à un niveau fixé).

Soit une instruction i_k, l'ensemble E_k des variables, ayant au moins une occurrence dans l'instruction i_k est noté : $E_k = \{x_1, x_2, ..., x_p\}$ (espace d'exécution de l'instruction i_k)

Environnement

C'est l'ensemble des objets et des structures nécessaires à l'exécution d'un travail donné pour un processeur fixé (niveau information).

Action

C'est l'opération ou le traitement déclenché par un événement qui modifie l'environnement (ou bien toute modification de l'environnement);

Action primitive

Pour un processeur donné(d'une machine abstraite d'un niveau fixé)une action est dite primitive, si l'énoncé de cette action est à lui seul suffisant pour que le processeur puisse l'exécuter sans autre éléments supplémentaires. Une action primitive est décrite par une *instruction simple* du processeur.

Action complexe

Pour un processeur donné(d'une machine abstraite d'un niveau fixé)une action complexe est une action **non-primitive**, qui est **décomposable** en actions primitives (à la fin de la phase de conception elle pourra être exprimée soit par un **module de traitement**, soit par une **instruction composée**).

Remarques :

- Ce qui est action primitive pour une machine abstraite de niveau n, peut devenir une action complexe pour une machine abstraite de niveau n+1, qui est l'expression de la précédente à un plus bas niveau (d'abstraction).
- Les instructions du langage doivent être les mêmes pour tous les niveaux de machine abstraite, sinon la programmation devient trop lourde à gérer.
- Tout langage de description de machine abstraite n'est pas implantable sur ordinateur (*au plus partiellement sinon ce serait tout simplement un langage de programmation*). Il ne peut servir qu'à décrire en partie la spécification et la conception. De plus il doit utiliser les idées de la programmation structurée descendante modulaire.

4. Un langage de description d'algorithme : LDFA

Avertissement

L'apprentissage d'un langage de programmation ne sert qu'aux phases 3 et 4 (traduction et exécution) et ne doit pas être confondu avec l'utilisation d'un langage algorithmique qui prépare le travail et n'est utilisé que comme plan de travail pour la phase de traduction. En utilisant la construction d'une maison comme analogie, il suffit de bien comprendre qu'avant de construire la maison, le chef de chantier a besoin du plan d'architecte de cette maison pour passer à la phase d'assemblage des matériaux ; il en est de même en programmation.

Enonçons un langage simple, extensible, qui est utilisé dans tout le reste du document et qui va servir à décrire les algorithmes. Nous le dénotons dans la suite du document comme **LDFA** pour **L**angage de **D**escription **F**ormel d'**A**lgorithme (terminologie non standard utilisée par l'auteur pour dénommer rapidement un langage algorithmique précis).

4.1 Atomes du LDFA

- Les ensembles de nombres comme **N**,**Z**,**Q**,**R** (les vrais ensembles classiques des mathématiques et leurs structures connues).
- La grammaire mathématique et celle du français.
- {**V**,**F**} comme éléments logiques (({**V**,**F**},$\neg$, $\wedge$, $\vee$) étant une algèbre de Boole)
- Les prédicats.
- Les caractères du français et les chaînes de caractères C des machines.

4.2 Information en LDFA

On rappelle qu'une information en LDFA est obtenue par le contenu d'un atome et se construit à l'aide de :

- la grammaire du français et le sens commun des mots,
- les théorèmes et les résultats obtenus des théories mathématiques(le sens étant le sens habituel donné à tous les symboles),
- toutes les manipulations générales (algorithmes en particulier) sur les structures de données.

4.3 Vocabulaire terminal du LDFA

V_T = **{** <-- , Ω , **lire( ) , ecrire( ) , si , tantque , alors , ftant , faire , fsi , sinon , sortirSi, pour , repeter , fpour , jusque , ; , entrée , sortie , Algorithme , local , global , principal , modules , specifications , types-abstraits , debut , fin , (,) , [,] , * , + , - , / ,** ¬ , ∧ , ∨ **}**

4.4 Instructions simples du LDFA :

Instruction vide

syntaxe : Ω

sémantique : ne rien faire pendant un temps de base du processeur.

Affectation

syntaxe : a <-- α
où : a∈ **identif, et** α est un schéma fonctionnel.

sémantique :
1) si α =identificateur alors val(a)=val(α)
2) si α est un atome alors val(a)=α
3) si α est une application / α : $(id_1,.....,id_p) \rightarrow \alpha(id_1,.....,id_p)$
alors val(a)= $\alpha'(val(id_1),.....,val(id_p))$
où α' est l'interprétation de α sur l'ensemble des valeurs des $val(id_k)$

Lecture

syntaxe : **lire**(a) (où a ∈ **identif)**

sémantique : le contexte de la phrase précise où l'on lit pour "remplir" a, sinon on indique **lire**(a) **dans** .....

Elle permet d'attribuer une valeur à un objet en allant lire sur un périphérique d'entrée et elle range cette valeur dans l'objet.

Ecriture

syntaxe : **ecrire**(a) (où a ∈ **identif**)

sémantique : le contexte de la phrase précise où l'on écrit pour "voir" a, sinon on indique **ecrire**(a) **dans** .....

Ordonne au processeur d'écrire sur un périphérique (Ecran, Imprimante, Port, Fichier etc...)

Condition

syntaxe : **si** P **alors** E1 **sinon** E2 **fsi**
où P est un prédicat ou proposition fonctionnelle,
E1 et E2 sont deux instructions composées.

sémantique : classique de l'instruction conditionnelle, si le processeur n'est pas lié au temps on peut écrire :

si P **alors** E1 **sinon** Ω **fsi** = **si** P **alors** E1 **fsi**

Nous notons = la relation d'équivalence entre instructions. Il s'agit d'une équivalence sémantique, ce qui signifie que les deux instructions donnent les mêmes résultats sur le même environnement.

Boucle tantque

syntaxe : **tantque** P **faire** E **ftant**
où P est un prédicat et E une instruction composée)

sémantique :
tantque P **faire** E **ftant** = **siP alors (E ; tantque P faire E ftant) fsi**

Remarques :

Au sujet de la relation "=" qui est la notation pour l'équivalence sémantique en LDFA, on considère un "programme" LDFA non pas comme une suite d'instructions, mais comme un environnement donné avec un état initial E0 , puis on évalue la modification de cet environnement que chaque action provoque sur lui:
$\{E_0\} \rightarrow \{E_1\} \rightarrow \{E_2\} \rightarrow ..... \rightarrow \{E_k\} \rightarrow \{E_{k+1}\}$

où action n+1 : $\{E_n\} \rightarrow \{E_{n+1}\}$.

On obtient ainsi une suite d'informations sur l'environnement :$(E_0, E_1, ..., E_{k+1})$

Nous dirons alors que deux instructions (simples ou composées) sont sémantiquement équivalentes (notation =)si leurs actions associées sur le même environnement de départ provoquent la même modification.

A chaque instruction est associée une action sur l'environnement, c'est le résultat qui est le même (même état de l'environnement avant et après) :
Soient : Instr1 → action1 (*action associée à Instr1*),
Instr2 → action2 (*action associée à Instr2*),

Soient E et E' deux états de l'environnement,
si nous avons : **{E} action1 {E'}** et **{E} action2 {E'}** ,alors Instr1 et Instr2 sont sémantiquement équivalentes, nous le noterons :

Instr1 = Instr2

Boucle répéter

syntaxe : **repeter** E **jusqua** P
(où P est un prédicat et E une instruction composée)

sémantique :
repeter E **jusqua** P = E ; **tantque** **not** P **faire** **E** **ftant**

Exemple d'équivalence entre itérations:

tantque P **faire** E **ftant** = **si P alors (repeter E jusqua not** P) **fsi**
repeter E **jusqua** P = E ; **tantque not** P **faire E ftant** *(par définition)*

Boucle pour

syntaxe : **pour x** <-- a **jusqua b faire** E **fpour**
(où E est une instruction composée, x une variable, a et b des expressions dans un ensemble fini F totalement ordonné, la relation d'ordre étant notée ≤ , le successeur d'un élément x dans l'ensemble est noté Succ(x) et son prédécesseur pred(x))

sémantiques :
Cette boucle fonctionne à la fois en suivant automatiquement l'ordre croissant dans l'ensemble fini F ou en suivant automatiquement l'ordre décroissant, cela dépendra de la position respective de départ de la borne a et de la borne b. La variable x est appelée un indice de boucle.

sémantique dans le cas ordre croissant à partir du tantque :

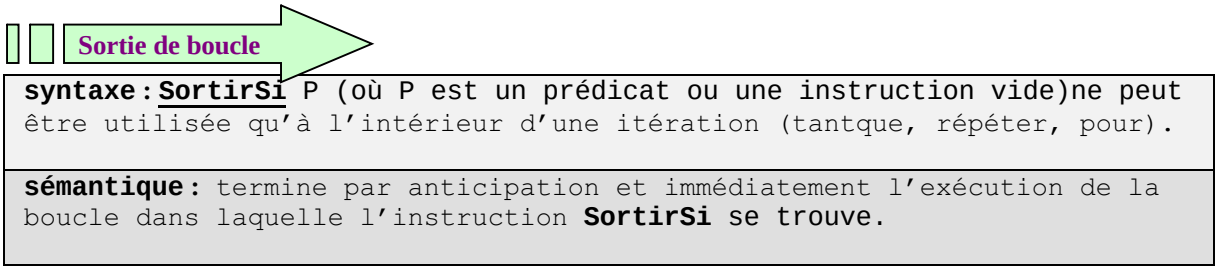
```
x <-- a ;
tantque x ≤ succ(b)faire
  E ;
  x <-- succ(x) ;
ftant
```

sémantique dans le cas ordre décroissant à partir du tantque :

```
x <-- a ;
tantque x ≥ pred(b) faire
  E ;
  x <-- pred(x) ;
ftant
```

Exemple simple :

- **E = N** *(entiers naturels)* et la relation d'ordre : ≤ = *inférieur ou égal dans* **N**
- **pour** i ≤ x **jusquà** y **faire** R **FinPour**
  (ici i prendra toutes les valeurs successives dans **N** comprises entre x et y soient, x+y-1 valeurs et s'incrémentera de 1 à chaque fois)



Exemple récapitulatif complet

Reprenons l'exemple précédent de l'équation du second degré en décrivant dans l'arbre de programmation l'action de la machine abstraite de chaque niveau à l'aide d'instructions du langage algorithmique LDFA :

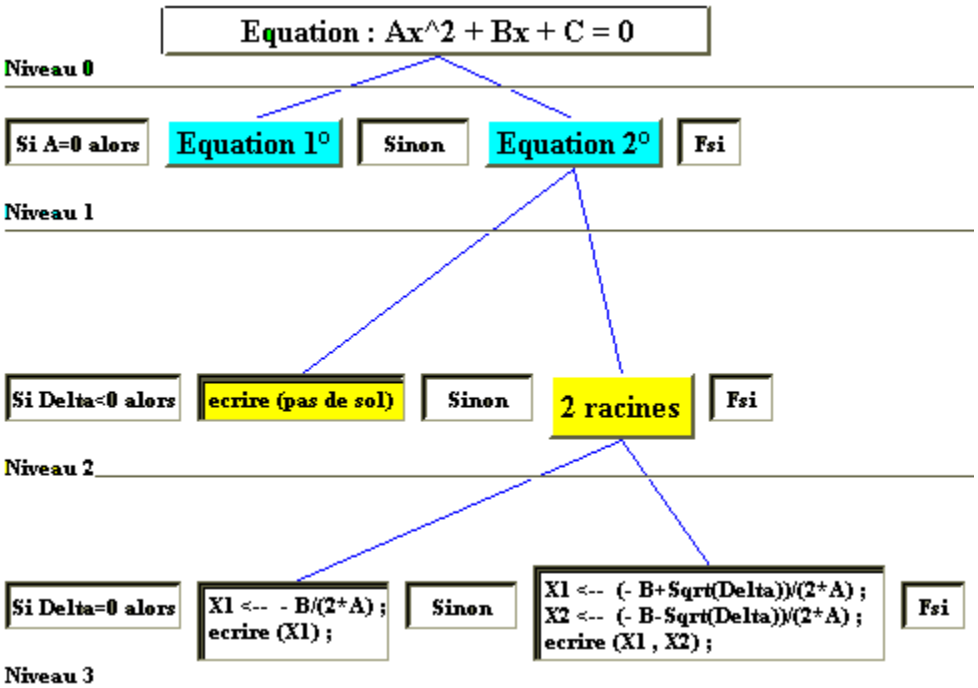


figure de la branche d'arbre 2ème degré

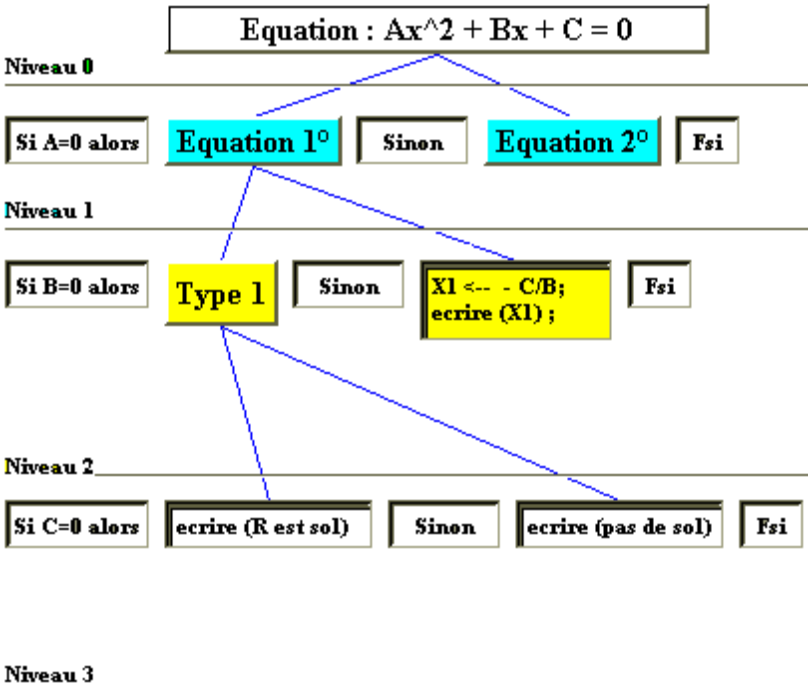


figure de la branche d'arbre 1er degré

Ecriture de l'algorithme

En relisant cet arbre selon un parcours en préordre *(il s'agit de parcourir l'arbre en partant de la racine et descendant toujours par le fils le plus à gauche, puis ensuite de passer au fils droit suivant etc…)*, l'on obtient après avoir complété l'algorithme une écriture linéaire comme suit :

```
Algorithme Equation
   Entrée: A,B,C ∈ R3
   Sortie: X1 ,X2 ∈ R2
   Local: Δ ∈ R
début
   lire(A,B,C);
   Si A=0 alors  { A=0 }
      Si B = 0 alors
         Si C = 0 alors
            écrire(R est solution)
         Sinon { C ≠ 0 }
            écrire(pas de solution)
         Fsi
      Sinon  { B ≠ 0 }
         X1 ← -C/B;
         écrire (X1)
      Fsi
   Sinon  { A ≠ 0 }
```

```
Sinon  { A ≠ 0 }
    Δ ← B² - 4*A*C ;
    Si Δ < 0 alors
          écrire(pas de solution)
    Sinon   { Δ ≥ 0 }
      Si Δ = 0 alors
         X1 ← -B / (2*A);
         écrire(X1)
      Sinon  { Δ > 0 }
         X1 ← (-B+sqrt(Δ)) / (2*A);
         X2 ← (-B-sqrt(Δ)) / (2*A);
         écrire( X1 , X2 )
      Fsi
    Fsi
Fsi
FinEquation
```

Nous regroupons toutes les informations de conception dans un document que nous appelons le dossier de développement.

5. Le Dossier de développement

C'est un document dans lequel se trouvent consignés tous les éléments relatifs à la construction et à l'écriture de l'algorithme et du programme résolvant le problème cherché. Nous le divisons en 5 parties.

5.1 Enoncé et spécification

Enoncé du problème résolu par ce logiciel.

- **Spécifications opérationnelles** des abstractions de plus haut niveau du logiciel, en exprimant celles-ci à l'aide de types abstraits et de spécifications de plus bas niveau.
- **Spécifications des types abstraits de données** utilisés.
- **Spécifications d'interface** pour les abstractions de plus bas niveau.

On utilisera ces trois techniques de spécification de manière descendante, quitte à remonter corriger des spécifications de niveau plus haut lorsque des erreurs seront apparues dans une spécification de plus bas niveau. Ces spécifications sont destinées au niveau " concepteur de logiciel ", plutôt qu'à l'utilisateur. Cette partie rassemble les définitions abstraites des composants. Un utilisateur de base n'ayant à priori pas à consulter ce paragraphe, les termes employés seront les plus rigoureux possibles relativement à un formalisme éventuel.

Analyse des besoins :
son utilité principale est de fournir à l'utilisateur la description des services que lui rendra ce logiciel. Les termes utilisés doivent être compris par l'utilisateur.

5.2 Méthodologie

Dans ce paragraphe se situent tous les documents et les explications qui ont pu mener à la décision de résoudre le problème posé par la méthode que l'on a choisie. Le programmeur dispose ici de toute latitude pour s'exprimer à l'aide de texte en langue naturelle, de représentation graphique, d'outils ou de supports permettant au lecteur de se faire une idée précise du pourquoi des choix effectués.

5.3 Environnement

L'étudiant pourra présenter sous forme d'un tableau les principales informations concernant les données de son algorithme.

Exemple :

Nom	**genre**	**localisation**	**utilisation**
PHT	reel	Entrée	prix hors taxe
TVA	reel	local	TVA en %
PTTC	reel	sortie	Prix TTC

5.4 Algorithme en LDFA

Ici se situe la description de l'algorithme proposé pour résoudre le problème proposé. Il est obtenu entre autre à partir de l'arbre de programmation construit pendant l'analyse et la conception. Ci-dessous le modèle général d'un algorithme :

Algorithme XYZT;
    global :
    local :
    entrée :
    sortie :
    modules utilisés :
Spécifications : (TAD)
  Types Abstraits de Données utilisés
début
  (corps d'algorithme en LDFA)
fin XYZT.

Nous verrons ailleurs ce que représentent les notions de TAD et de module.

5.5 Programme en langage de programmation *(Pascal par exemple)*

Dans ce paragraphe nous ferons figurer la " traduction " en langage de programmation de l'algorithme du paragraphe précédent.

6. Trace formelle d'un algorithme

Et le dernier [précept], de faire partout des dénombrements si entiers, et des revues si générales, que je fusse assuré de ne rien omettre.
R Descartes Discours de la méthode, seconde partie, 1637.

Nous proposons au débutant de vérifier l'exactitude de certaines parties de son algorithme en utilisant un petit outil permettant l'exécution formelle (c'est à dire sur des valeurs algébriques ou symboliques plutôt que numériques) de son algorithme. La trace numérique et les vérifications associées seront effectuées lors de l'exécution par la machine.

6.1 Espace d'exécution d'une instruction composée

On appelle espace d'exécution d'une séquence ou d'un bloc d'instructions $i_1 ... i_n$ l'ensemble $E = \bigcup_{k=1}^{n} E_k$ où E_k est l'espace d'exécution de l'instruction i_k.

Rappelons que l'on peut considérer un " programme " LDFA sous un autre point de vue : non pas comme une suite d'instructions, mais comme un environnement donné avec un état initial E_0 , puis on évalue la modification de cet environnement que chaque instruction provoque sur lui.

On considère les instructions i_k comme des transformateurs d'environnement E_n:
$\{E_0\} \rightarrow \{E_1\} \rightarrow \{E_2\} \rightarrow ..... \rightarrow \{E_k\} \rightarrow \{E_{k+1}\}$

L'instruction i_{n+1} fait alors passer l'environnement de l'état E_n à l'état E_{n+1}.
Nous écrirons ainsi : $i_{n+1} : \{E_n\} \rightarrow \{E_{n+1}\}$

Ces actions déterminent alors une suite d'états de l'environnement $(E_0,E_1,....E_{k+1})$ que l'on peut observer.

C'est ce point de vue qui permet d'exécuter un suivi d'exécution symbolique d'un algorithme. Nous le nommerons " trace formelle ".

On adoptera pour une trace formelle une disposition en tableau de l'espace d'exécution comme suit :

Etats	V_1	V_2	.....	V_n
E_1	--	--		**y**
E_2	**x**	--		**y+1**

La colonne Etats représente donc les états successifs de l'environnement (ou espace d'exécution) figuré ici par les variables $V_1,V_2,...,V_n$. Les contenus des cellules du tableau sont les valeurs symboliques des variables au cours du déroulement de l'exécution. On peut considérer l'image mentale suivante de la trace formelle comme étant une succession de "photographies instantanées" de l'environnement prises après chaque instruction.

6.2 Exemple complet avec trace formelle

Nous traitons un exemple complet avec son dossier de développement et une trace formelle.

Enoncé

Calculer $S=\sum_{i=1}^{n} i$ sans utiliser de formule (*car l'on sait que S=(n+1)n/2*)

Spécification : flux d'information

En Entrée	**En Sortie**
Un nombre $n \in N^*$	Ecrire la somme voulue S.

Méthodologie

S est la somme des termes d'une suite récurrente :

$$s_i \begin{cases} s_0 = 0 \\ s_i = s_{i-1} + i \end{cases}$$

Environnement

Nom	**genre**	**localisation**	**utilisation**
N	Entier	Entrée	Nombre d'éléments à saisir
S	Entier	Sortie	Variable de cumul pour la somme
I	Entier	local	Gestion des boucles : compteur

Algorithme

```
Algorithme Somentier
| N ∈ N*
| S , I ∈ N²
|
Début {Somentier}
| (E0)
| Lire (N) ;
| (E1)
| S ← 0;
| (E2)
| I ← 1;
| (E3)
| TantQue I ≤ ? ? ? faire
| |   (E4)
| |   S ← S+I;
| |   (E5)
| |   I ← I+1;
| |   (E6)
| FinTant;
| (E7)
| Ecrire(S);
Fin Somentier
```

Ceci est un algorithme incomplet dans lequel on a déjà intercalé les états (E_n) entre les instructions. On ne sait pas exactement quel sera le test d'arrêt de la boucle (remplacé par ? ? ?), on sait seulement que c'est la valeur de la variable de compteur **I** qui le fournira.

Utilisation de la trace formelle

Nous allons montrer à l'aide de la trace formelle que cet algorithme fournit bien la somme des n premiers entiers dans la variable S, relativement aux préconditions { S = 0 et i = 1}.

Nous allons donc faire de la démonstration de programme :

Précondition	Action	Postcondition
{S= 0 et i= 1}	**Algorithme**	$\{ S = \sum_{i=1}^{n} i \}$

Nous pouvons avoir une hésitation quant à la borne du test "**TantQue** I ≤ ??? ", faut-il s'arrêter à N, N-1 ou N+1 ?

Posons comme hypothèse que le test s'arrête à la valeur N, soit : "**TantQue** I ≤ N …

Exécutons manuellement et pas à pas l'algorithme précédent en supposant que le test d'arrêt n'est pas franchi, c'est à dire que l'on a I > N.

Voici le début des résultats de sa trace formelle dans le tableau ci-dessous :

Etats	I	N	S
E_0	-	-	--
E_1	-	**n**	--
E_2	-	**n**	**0**
E_3	**1**	**n**	**0**
$E_4=E_3$	**1**	**n**	**0**
E_5	**1**	**n**	**1**
E_6	**2**	**n**	**1**
$E_4=E_6$	**2**	**n**	**1**
E_5	**2**	**n**	**3**
E_6	**3**	**n**	**3**
$E_4 = E_6$ etc..	**3**	**n**	**3**

isolons les deux premiers " tours " de boucle :

$E_4 = E_6$	**2**	**n**	**1**
$E_4 = E_6$…	**3**	**n**	**3**

Nous voyons que juste avant la sortie de boucle (état E_6) au premier tour I=2 et S=1, au deuxième tour I=3 et S=3 .

Nous posons l'hypothèse de récurrence qu'au kème tour i=k+1 et S= $\sum_{i=1}^{k} i$ (somme des k premiers entiers). Nous allons utiliser l'exécution formelle pas à pas d'un tour de boucle afin de voir si après un tour de plus cette hypothèse se vérifie au rang k+1 :

Etats	I	N	S
.....	**...**	**...**	**...**
$E_4 = E_6$	**k+1**	**n**	**S=** $\sum_{i=1}^{k} i$
E_5	**k+1**	**n**	**S=** $\sum_{i=1}^{k} i$ **+ k+1**
E_6	**k+2**	**n**	**S=** $\sum_{i=1}^{k} i$ **+ k+1**

Or S= $\sum_{i=1}^{k} i$ +k+1 = $\sum_{i=1}^{k+1} i$ (la somme des k+1 premiers entiers).

Nous venons donc de montrer qu'à l'état E6 cet algorithme donne :

Etats	I	N	S
E_6	k+1	n	$S=\sum_{i=1}^{k} i$

En particulier, lorsque k = n nous avons dans S la somme des n premiers entiers $\sum_{i=1}^{n} i$:

Etats	I	N	S
E_6	n+1	n	$S=\sum_{i=1}^{n} i$

Nous pouvons déjà écrire que : $\forall$ n, n > 0, $S = \sum_{i=1}^{n} i$

En plus ce dernier tableau nous permet immédiatement de trouver la valeur exacte de la variable de contrôle de la boucle (ici la variable I qui vaut n+1) et donc d'écrire un test d'arrêt de boucle juste.

On peut alors choisir comme test I<>n+1 ou bien I< n+1 etc... ou tout autre prédicat équivalent.

Il était possible de programmer directement cet algorithme avec les deux autres boucles (pour... et répeter...). Ceci est proposé en exercice au lecteur.

7. Traducteur élémentaire LDFA - Java / Pascal

Nous venons de voir qu'un algorithme devait se traduire en langage de programmation (dit évolué). Nous fournirons ici un tableau qui sera utile à l'étudiant pour la traduction des instructions algorithmiques en langage de programmation.

7.1 Traducteur

Afin de bien montrer que l'écriture algorithmique est plus abstraite qu'un langage de programmation nous donnons un tableau de traduction LDFA dans deux langages : en Pascal de base et en Java2 restreint aux instructions seulement: (dans le tableau, P est un prédicat et E une instruction composée)

LDFA	Java	Pascal
Ω (instruction vide)	pas de traduction	pas de traduction
debut i1 ; i2; i3; ..... ; ik **fin**	**{** i1 ; i2; i3; ..... ; ik **}**	**begin** i1 ; i2; i3; ..... ; ik **end**
x ← a	x = a ;	x := a
;	pas de traduction	(ordre d'exécution) ;
Si P **alors** E1 **sinon** E2 **Fsi**	**if** (P) E1 ; **else** E2 ; *(attention, pas de fermeture !)*	**if** P **then** E1 **else** E2 *(attention, pas de fermeture !)*
Tantque P **faire** E **Ftant**	**while** (P) E ; *(attention, pas de fermeture)*	**while** P **do** E *(attention, pas de fermeture)*
répeter E **jusquà** P	**do** E ; **while** (! P) ;	**repeat** E **until** P
lire (x1,x2,x3.....,xn)	System.in.read( ) ; System.in.readln( ) ;	read(fichier,x1,x2,x3.....,xn) readln(x1,x2,x3.....,xn) Get(fichier)
ecrire (x1,x2,x3.....,xn)	System.in.print( ) ; System.in. println( ) ;	write(fichier,x1,x2,x3.....,xn) writeln(x1,x2,x3.....,xn) Put(fichier)
pour x ← a **jusquà** b **faire** E **Fpour**	**for** (**int** x= a; x <= b; x++) E ;	**for** x:=a **to** b **do** E *(croissant)* **for** x:=a **downto** b **do** E *(décroissant)* *(attention, pas de fermeture)*
SortirSi P	**if** (P) break **;**	**if** P **then** Break

Ce tableau de traduction permet déjà d'écrire très rapidement des programmes Pascal et Java simples à partir d'algorithmes étudiés et écrits.

7.2 Exemple

En appliquant le traducteur à l'algorithme de l'équation du second degré nous obtenons le programme Pascal suivant :

```
program equation;
var
   A,B,C:real;
   X1,X2:real;
   Delta:real;
begin
   readln(A,B,C);
   if A = 0 then {A=0}
      if B = 0 then
        if C = 0 then
           writeln('R est solution')
        else
            writeln('pas de solution')
```

```
    else
    begin
      X1 := - C/B;
      writeln('x=',X1)
    end
  else
  begin
    Delta := B*B-4*A*C;
    if Delta < 0 then
        writeln('pas de solution')
    else
     if Delta=0 then
     begin
       X1 := -B/(2*A);
       writeln('x=',X1)
     end
     else
     begin
       X1 := (-B + Sqrt(Delta)) / (2*A);
       X2 := (-B - Sqrt(Delta)) / (2*A);
       writeln('x1=',X1,'x2=',X2)
     end
  end
end.
```

En appliquant le traducteur Java2 à ce même algorithme de l'équation du second degré, nous obtenons le squelette de programme Java2 suivant :

```
if (a ==0)
  if (b ==0)
    if (c ==0)
      System.out.println("tout reel est solution") ;
    else
      System.out.println("il n'y a pas de solution") ;
  else  {
    x = -c/b ;
    System.out.println("la solution est " + x) ;
  }
else  {
  delta = b*b -4*a*c ;
  if (delta <0)
    System.out.println("il n'y a pas de solution dans les reels") ;
  else
    if (delta == 0)  {
        x1 = -b / (2*a) ;
        System.out.println("il y a une solution double : "+x1) ;
    }
    else  {
      x1 = (-b + Math.sqrt(delta)) / (2*a) ;
      x2 = (-b - Math.sqrt(delta)) / (2*a) ;
      System.out.println("il y deux solutions égales a "+x1+" et " + x2) ;
    }
} etc…
```

7.3 Sécurité et ergonomie

L'utilisation du traducteur manuel LDFA ––> Pascal fournit une version préliminaire de programme pascal fonctionnant sur des données correctes sans aucune présentation.

Il appartient au programmeur de compléter dans un deuxième temps la partie sécurité associée aux contraintes du domaine de définition des variables et aux contraintes matérielles d'implantation. Enfin, dans un troisième temps, l'ergonomie (forme de l'échange d'information entre le programme et le futur utilisateur) sera envisagée et programmée.

Voyons sur l'exemple de la somme des n premiers entiers déjà cité plus haut, comment ces trois étapes s'articulent .

Etape de traduction-*somme des n premiers entiers*

Texte final de l'algorithme de départ :	Texte de sa traduction en pascal :

Algorithme Somentier
N $\in$ **N***
S , I $\in$ **N**^{2}
Début *{Somentier}*
Lire (N) ;
S $\leftarrow$ 0;
I $\leftarrow$ 1;
TantQue I < N+1 **faire**
S $\leftarrow$ S+I;
I $\leftarrow$ I+1;
FinTant;
Ecrire(S);
Fin Somentier
	program Somentier ;
	var N : integer ;
	S,I : integer ;
	begin
	readln(N) ;
	S :=0 ;
	I :=1 ;
	while I < N+1 **do**
	begin
	S := S +I;
	I := I+1;
	end;
	writeln(S)
	end.

Etape de sécurisation-*somme des n premiers entiers*

Sécurité due aux domaines de définition des données

La traduction ne permet pas d'écrire les domaines de définition des variables : en l'occurrence ici la variable N$\in$**N*** est traduite par " **var** N : **integer** ", or le type prédéfini **integer** est un sous-ensemble des entiers relatifs **Z**, il est donc nécessaire d'éliminer les entiers négatifs ou nuls comme choix possible.

Dès que l'utilisateur aura entré son nombre, le programme devra tester l'appartenance au bon intervalle afin de protéger la partie de code, par exemple avec une instruction de condition :

```
if  N > 0  then begin

// code protégé
end
```

Protection programmée dans les pointillés en dessous dans le cadre de droite :

```
program Somentier ;
var N : integer ;
    S,I : integer ;
begin
readln(N) ;

S :=0 ;
I :=1 ;
while I < N+1 dobegin
  S := S + I;
  I := I+1;
end;
writeln(S)

end.
```

```
program Somentier ;
var N : integer ;
    S,I : integer ;
begin
readln(N) ;
if  N > 0  then begin
    S :=0 ;
    I :=1 ;
    while I < N+1 do begin
      S := S + I;
      I := I+1;
    end;
    writeln(S)
 end
end.
```

Sécurité due aux contraintes d'implantation

Si nous exécutons ce programme pour la valeur N=500, la valeur fournie en sortie est " -5822 " sur un pascal 16 bits comme TP-pascal, le résultat n'est pas correct. Nous sommes confrontés au problème de la représentation des entiers machines déjà cité. Ici le type **integer** est restreint à l'intervalle *[-32768,+32767]* ; il y a manifestement dépassement de capacité (overflow) et le système a allègrement continué les calculs malgré ce dépassement. En effet, la somme vaut 500*501/2 soit **125250**, cette valeur n'appartient pas à l'intervalle des **integer**.

Le programmeur doit donc remédier à ce problème par un effort personnel de sécurisation de son programme en n'autorisant les calculs que pour des valeurs valides offrant un maximum de sécurité.

Ici la variable S contient la somme $\sum_{i=1}^{k} i$, nous savons que $\sum_{i=1}^{k} i$ = k(k+1)/2, donc il suffira de résoudre dans **N** l'inéquation k(k+1)/2 $\leq$ 32767 où n est l'inconnue. L'unique solution positive a pour partie entière 255, qui est la valeur maximale avant dépassement de capacité. Donc il suffit de protéger le code par un test supplémentaire sur la variable N :

```
if (N > 0) and (N < 256) then begin
  S :=0 ;
  I :=1 ;
  while I< N+1 do begin
      S := S +I;
      I := I+1;
   end;
   writeln(S)
end
```

En vérifiant sur l'exécution, nous trouvons que S = 32640 pour N = 255. Ce qui nous donne la version suivante du programme :

```
program  Somentier ;
var N : integer ;
      S , I : integer ;
begin
   readln(N) ;
   if (N > 0) and (N < 256) then begin
         S :=0 ;
         I :=1 ;
         while I< N+1 do begin
            S := S +I;
            I := I+1;
         end;
         writeln(S)
   end
end.
```

Etape d'ergonomie-*somme des n premiers entiers*

Dans cet exemple, l'information à échanger avec l'utilisateur est très simple et ne nécessite pas une interface spéciale. Il s'agira de lui préciser les contraintes d'entrée et de lui présenter d'une manière claire le résultat.

```
program  Somentier ;
var N : integer ;
      S , I : integer ;
begin
   Write('Entrez un entier entre 0 et 255') ;
   readln(N) ;
   if (N > 0) and (N < 256) then begin
         S :=0 ;
         I :=1 ;
         while I< N+1 do begin
            S := S +I;
            I := I+1;
         end;
         writeln('la somme des ',N,' premiers entiers vaut ',S)
   end
   else
      writeln('Calcul impossible ! !')
end.
```

Vous remarquerez que les adjonctions supplémentaires de code *(en italique)* dans le programme final se montent à environ 50% du total du code écrit, car un logiciel n'est pas uniquement un algorithme traduit.

En continuant d'appliquer le principe de la programmation structurée, il est bon de bien séparer lors du développement la partie algorithmique des parties sécurité et ergonomie. Le programmeur débutant y gagnera en clarté dans sa méthode de travail.

8. Facteurs de qualité du logiciel

B.Meyer et G.Booch

Constat

Un utilisateur, lorsqu'il achète un produit comme un appareil électro- ménager ou une voiture, attend de son acquisition qu'elle possède un certain nombre de **qualités** (fiabilité, durabilité, efficacité, ...). Il en est de même avec un logiciel.

Voici une liste minimale de critères de qualité du logiciel (proposée B.Meyer, G.Booch):

Correction	**Robustesse**	**Extensibilité**
Réutilisabilité	**Compatibilité**	**Efficacité**
Portabilité	**Vérificabilité**	**Intégrité**
Facilité utilisation	**Modularité**	**Lisibilité**
Abstraction		

Reprenons les définitions communément admises par ces deux auteurs sur ces facteurs de qualité.

Correction — La **correction** est la qualité qu'un logiciel a de respecter les spécifications qui ont été posées.

Robustesse — La **robustesse** est la qualité qu'un logiciel a de fonctionner en se protégeant des conditions de dysfonctionnement.

Extensibilité — **L'extensibilité** est la qualité qu'un logiciel a d'accepter des modifications dans les spécifications et des adjonctions nouvelles.

Réutilisabilité — La **réutilisabilité** est la qualité qu'un logiciel a de pouvoir être intégré totalement ou partiellement sans réécriture dans un nouveau code.

Compatibilité — La **compatibilité** est la qualité qu'un logiciel a de pouvoir être utilisé avec d'autres logiciels sans autre effort de conversion des données par exemple.

Efficacité — **L'efficacité** est la qualité qu'un logiciel a de bien utiliser les ressources.

Portabilité — La **portabilité** est la qualité qu'un logiciel a d'être facilement transféré sur de nombreux matériels, et insérable dans des environnements logiciels différents.

Vérificabilité — La **vérificabilité** est la qualité qu'un logiciel a de se plier à la détection des fautes, au traçage pendant les phases de validation et de test.

Intégrité — **L'intégrité** est la qualité qu'un logiciel a de protéger son code et ses données contre des accès non prévus.

Facilité — La **facilité** d'utilisation est la qualité qu'un logiciel a de pouvoir être appris, utilisé, interfacé, de voir ses résultats rapidement compris, de pouvoir récupérer des erreurs courantes.

Lisibilité — La **lisibilité** est la qualité qu'un logiciel a d'être lu par un être humain.

Modularité — La **modularité** est la qualité qu'un logiciel a d'être décomposable en éléments indépendants les uns des autres et répondants à un certain nombre de critères et de principes.

Abstraction — **L'abstraction** est la qualité qu'un logiciel a de s'attacher à décrire les opérations sur les données et à ne manipuler ces données qu'à travers ces opérations.

La production de logiciels de qualité n'est pas une spécificité des professionnels de la programmation ; c'est un état d'esprit induit par les méthodes du génie logiciel. Le débutant peut, et nous le verrons par la suite, construire des logiciels ayant des " qualités " sans avoir à fournir d'efforts supplémentaires.

Bien au contraire la réalité a montré que les étudiants " bricoleurs " passaient finalement plus de temps à " bidouiller " un programme que lorsqu'ils décidaient d'user de méthode de travail. Une amélioration de la qualité générale du logiciel en est toujours le résultat.

Machines abstraites : exemple

Traitement descendant modulaire d'un exemple complet

Objectif : développer un exemple simple de construction d'une machine abstraite par décomposition descendante sur 4 niveaux.

ENONCE

On donne une liste de n noms (composés de lettres uniquement). Extrayez ceux qui sont le premier et le dernier par ordre alphabétique. Ecrire un programme Pascal effectuant cette opération.

SPECIFICATIONS :(il s'agit d'éclaircir certaines décisions)

Plan:

Objets utilisés,
machine abstraite,
spécification de données.

Objets utilisés au niveau 1

Identification	Signification
(**Liste** , <<)	**Liste** est un ensemble fini de noms où << est une relation d'ordre total
Noms	Ensemble de tous les noms possibles (chacun est constitué de lettres)
élément	Fonction fournissant le kième élément de la liste : **élément : N*** x Liste → Liste
Grand	Un élément de l'ensemble ***Noms*** : Grand ∈ ***Noms***
Petit	Un élément de l'ensemble ***Noms*** : Petit ∈ ***Noms***
Long	Fonction fournissant le nombre d'éléments de la liste : Long : Liste → **N***

Machine abstraite de niveau 1

(description de haut niveau d'abstraction de l'algorithme choisi)

```
Grand ← élément ( 1 , Liste) ;
Petit  ← élément ( 1 , Liste) ;
Pour indice  ← 2 jusquà Long (Liste) faire
  Si Grand << élément (indice, Liste) alors
      Grand  ← élément (indice, Liste)
  fsi ;
  Si élément (indice ,Liste) << Petit alors
       Petit  ←  élément (indice, Liste)
  fsi
Fpour ;
```

{Grand = le dernier et Petit = le premier }

Les données au niveau 1

Identification	Signification
Noms	Un ensemble de caractères
(**Liste** , <<)	**Liste** est un ensemble fini muni d'une relation d'ordre total <<
élément	Fonction **élément : N*** x Liste → Liste (k, Liste) → **élément** (k, Liste) ∈ Liste
Long	Fonction fournissant le nombre d'éléments de la liste : Long : Liste → **N*** Liste → Long (Liste) = n

Nous avons ici une spécification abstraite de haut niveau. Il est impératif de prendre des décisions sur les structures de données qui vont être utilisées. Nous allons envisager le cas le plus simple : celui où la structure choisie pour représenter la liste est un tableau.

Les données au niveau 2

Reprise des objets abstraits en les exprimant de la façon suivante :

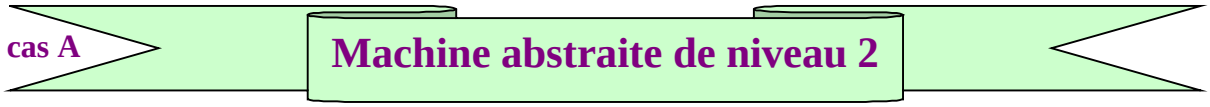
Cas A où la version pascal contient déjà les outils de chaînes

- Liste = Tableau
- élément (i, Liste) = Liste[i]
- Long(Liste) = n , taille du tableau
- ***Noms*** : Type string
- << : ≤ (relation de comparaison lexicographique sur les chaînes)

Nous continuons à descendre dans les niveaux d'abstraction. Nous devons prendre des décisions sur le langage-cible. Il est dit dans l'énoncé que ce doit être Pascal, mais lequel ?

Nous avons choisi dans ce premier cas, une version simple en prenant par exemple comme dialecte deux descendants de l'UCSD-Pascal, à savoir Think Pascal (Mac) ou Borland Pascal-Delphi (Windows-Linux) qui contiennent en prédéfini le type de chaîne de caractères.

Ces spécifications de données étant établies, la machine précédente devient :



Algorithme EXTRAIT0
Global : **n** ∈ **N*** , Long_mot ∈ **N***
Local : indice ∈ **N***, (Grand , Petit) ∈ **Noms** ^{2}
Spécification:
Noms = Type **string** prédéfini par une version d'implémentation du pascal.
Liste = Tableau de Nom, de Taille **n** ∈ **N*** fixée.
Début

Grand ← Liste[1] ;
Petit ← Liste[1] ;
Pour indice ←2 **jusquà** n **faire**
Si Grand < Liste[indice] **alors** Grand ← Liste[indice] **fsi** ;
Si Liste[indice] < Petit **alors** Petit ← Liste[indice] **fsi**
Fpour ;

Fin_EXTRAIT0

Cet algorithme se traduit immédiatement en pascal. Nous voyons donc qu'il nous a été possible d'écrire ce petit programme en descendant uniquement sur 2 niveaux de spécifications.

Qu'en est-il lorsque l'on travaille avec un autre langage cible (nous allons juste utiliser une version différente du même langage cible) ? Nous allons voir que nous devrons descendre alors plus loin dans les spécifications et développer plus de code c'est l'objectif de la suite de ce document.

Cas B où la version pascal ne contient pas les outils de chaînes

Reprenons partiellement la même spécification de données par un tableau de taille n pour la Liste, mais supposons que le langage-cible soit du **Pascal ISO**, dans lequel le type string n'existe pas.

cas B

Les données au niveau 2

Cas B où la version pascal ne contient pas d'outils de chaînes

- ***Noms*** : Nous choisissons, afin de ne pas nous perdre en complexités inutiles, de spécifier la *l'ensemble des caractères* par *un tableau de caractères :* notons le **Tchar**
- Liste = Tableau de **Tchar**
- élément (i, Liste) = Liste[i]
- Long(Liste) = n , taille du tableau .
- << : CMP (opérateur de relation de comparaison sur les **Tchar**)

cas B

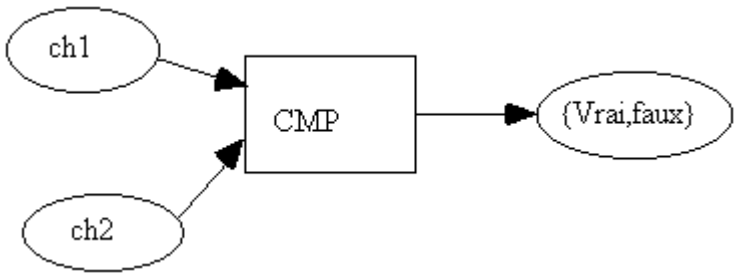
Machine abstraite CMP

Ces choix de spécification induisent un choix de développement d'une machine abstraite spécifique à la relation d'ordre << , qui n'est pas un opérateur simple du langage : nous avons dénoté la machine par le nom CMP.

Noms = Tchar
Taille de Tchar = n
ch1 $\in$ Tchar , ch2 $\in$ Tchar

Spécification de l'opérateur 'CMP 'de comparaison de chaînes :

CMP : **Tchar** x **Tchar** $\rightarrow$ { **Vrai** , **Faux** }
CMP (ch1,ch2) = **Vrai** **ssi** (ch1 < ch2) ou (ch1 = ch2)
CMP (ch1,ch2) = **Faux** **ssi** ch2 < ch1



ch1 $\in$ ***Noms***,
ch2 $\in$ ***Noms***

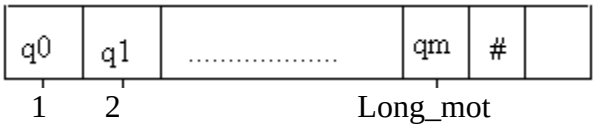
Descendons dans les spécifications plus concrètes de la machine abstraite de comparaison, spécifions d'une manière plus détaillée, les données **Noms** et **Liste** de la machine abstraite CMP, en répondant aux deux questions ci-dessous :

Noms = Tableau de caractères noté **Tchar** , comment est-il représenté ?
Liste = Tableau de **Tchar** , comment est-il représenté ?

cas B

Les données au niveau 3

Un nom **Noms** :



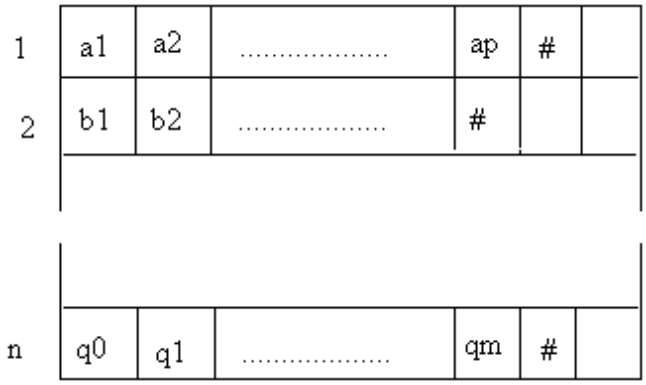
Noms = Tableau de caractères

Noms [i] = le caractère de rang i-1

Attributs :

- **Taille = Long_mot**
- **caractère spécial = #**

Une liste de noms **Liste** :



Liste = Tableau de noms

Liste[i] = le **Noms** de rang i

Attribut :

Taille = n

On dispose d'une relation d'ordre sur les caractères (ordre ASCII) notée ≤ .

cas B

Machine CMP de niveau 3

Décrivons une première version de CMP en tenant compte des spécifications de données précédentes.

Tantque (les caractères lus de ch1 et de ch2 sont les mêmes)
 et (ch1 non entièrement exploré) **et** (ch2 non entièrement exploré) **faire**
 passer au caractère lu suivant dans ch1 ;
 passer au caractère lu suivant dans ch2 ;
Ftant ;
Si (ch1 et ch2 finis en même temps) **alors** ch1=ch2 **fsi** ;
Si (ch1 fini avant ch2) **ou** (car_Lu de ch1 < car_Lu de ch2) **alors** ch1 < ch2 **fsi** ;
Si (ch2 fini avant ch1) **ou** (car_Lu de ch2 < car_Lu de ch1) **alors** ch2 < ch1 **fsi** ;

Descendons plus bas dans les niveaux d'abstraction.

cas B

Machine CMP de niveau 4

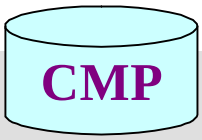
Notre travail va consister à expliciter, à l'aide des structures de données choisies les phrases de haut niveau d'abstraction de la spécification de niveau 3 de CMP (en italique le niveau 3, en gras le niveau 4):

spécification de niveau 3 de CMP	spécification de niveau 4 de CMP
car_Lu de ch1	**ch1[i]** (ième caractère de ch1)
car_Lu de ch2	**ch2[k]** (kème caractère de ch2)
ch1 fini ou entièrement exploré	**ch1[i] = #**
Ch2 fini ou entièrement exploré	**ch2[k] = #**
les caractères lus de ch1 et de ch2 sont les mêmes	**ch1[i] = ch2[i]**
caractère suivant de ch1 , ch2	Si caractère actuel = **ch1[k]** alors caractère suivant = **ch1[k+1] ,** **idem pour ch2**

La spécification opérationnelle de niveau 4 de CMP devient alors :

Tantque (ch1[k] = ch2[k]) et (ch1[k] ≠ #) et (ch2[k] ≠ #) **faire**
  k ← k+1
Ftant ;
Si (ch1[k] = #) **et** (ch2[k] = #) **alors** CMP ← Vrai **fsi** ;
Si (ch1[k] = #) **ou** (ch1[k] < ch2[k]) **alors** CMP ← Vrai **fsi** ;
Si (ch2[k] = #) **ou** (ch2[k] < ch1[k]) **alors** CMP ← Faux **fsi** ;

Il faut prévoir d'initialiser le processus au premier caractère k=1 d'où maintenant une spécification de l'algorithme :



Algorithme CMP
Local : k ∈ **N***
entrée : (ch1 , ch2) ∈ **Noms**^{2}
sortie : CMP ∈ { Vrai, Faux }
Spécification:
Noms = Tableau de Taille Long_mot fixée disposant d'un caractère de fin (#)
Début
 k ← 1 ;
 Tantque(ch1[k] = ch2[k]) et (ch1[k] ≠ #) et (ch2[k] ≠ #) **faire**
   k ← k+1
 Ftant ;
 Si(ch1[k] = '#') **et** (ch2[k] = '#') **alors** CMP ← Vrai **fsi** ;
 Si (ch1[k] = '#') **ou** (ch1[k] < ch2[k]) **alors** CMP ← Vrai **fsi** ;
 Si (ch2[k] = '#') **ou** (ch2[k] < ch1[k]) **alors** CMP ← Faux **fsi** ;
Fin_CMP.

Puis en intégrant la machine abstraite CMP de niveau 4 avec les spécifications de TAD décrites précédemment, le tout dans la spécification de niveau 3 de l'algorithme choisi, nous obtenons l'algorithme final suivant :

cas B **Algorithme** EXTRAIT1 **niveau 4**

Algorithme EXTRAIT1
Global : n $\in$ **N***, Long_mot $\in$ **N***
Local : indice $\in$ **N***, (Grand , Petit) $\in$ **Noms**^{2}

module utilisé : **CMP**

Spécification:
Noms = Tableau de caractères, de Taille Long_mot fixée disposant d'un attribut marqueur de fin, qui est le caractère spécial # .
Liste = Tableau de *Noms*, de Taille **n** $\in$ **N*** fixée.

TAD utilisés:
- Tableau de caractère de dimension 1.
- Tableau de *Noms* de dimension 1.

Début

Grand ← Liste[1] ;
Petit ← Liste[1] ;
Pour indice ← 2 **jusquà** n **faire**
Si CMP(Grand , Liste[indice]) **alors** Grand ← Liste[indice] **fsi** ;
Si CMP(Liste[indice] , Petit) **alors** Petit ← Liste[indice] **fsi**
Fpour ;

Fin_EXTRAIT1.

Voici une traduction possible en Pascal de cet algorithme.

```
Program EXTRAIT1;

Const
 Taille = 5;
 Long_mot = 20;
Type
   Nom = array[1..Taille] of Char;
   List_noms = array[1..Long_mot ] of Nom;
Var
   Liste : List_noms;
   indice : integer;
   Grand,Petit : Nom;
```

```
function CMP (ch1,ch2:Nom):Boolean;
var
  k : integer;
begin
   k:=1
   While (ch1[k]=ch2[k])and(ch1[k]<'#')and(ch2[k]<'#') do k:=k+1;
   if (ch1[k]='#')and(ch2[k]='#') then result :=True;
   if (ch1[k]='#')or(ch1[k]<ch2[k]) then result :=True;
   if (ch2[k]='#')or(ch2[k]<ch1[k]) then result :=False;
end;{CMP}

procedure INIT_Liste;
begin
      {initialise la liste des noms terminés par des #}
end;

procedure ECRIRE_Nom (name:Nom);
begin
     {écrit sur une même ligne les caractères qui composent la variable name, sans le #}
end;{ECRIRE_Nom}

Begin{EXTRAIT}
   INIT_Liste;
   Grand:=Liste[1];
   Petit:=Liste[1];
   for indice:=2 to taille do
   begin
     if CMP(Liste[indice],Petit) then Petit := Liste[indice];
     if CMP(Grand,Liste[indice]) then Grand := Liste[indice];
   end;
   write('Le premier est : ');
   ECRIRE_Nom(Petit);
   write('Le dernier est : ');
   ECRIRE_Nom(Grand);
End.{ EXTRAIT }
```

Le lecteur comprendra à partir de cet exemple que les langages de programmation sont très nombreux et que le choix d'un langage pour développer la solution d'un problème est un élément important.

Autres versions possibles à partir de CMP

La version d'implantation de CMP du niveau 4' a été conçue sur une structure de données tableau terminée par une sentinelle (le caractère #). Elle a été implantée par une fonction en pascal.

Il est possible de réécrire d'autres version d'implantation de cette même machine CMP avec des structures de données différentes comme un tableau avec un attribut de longueur ou bien une structure liste dynamique :

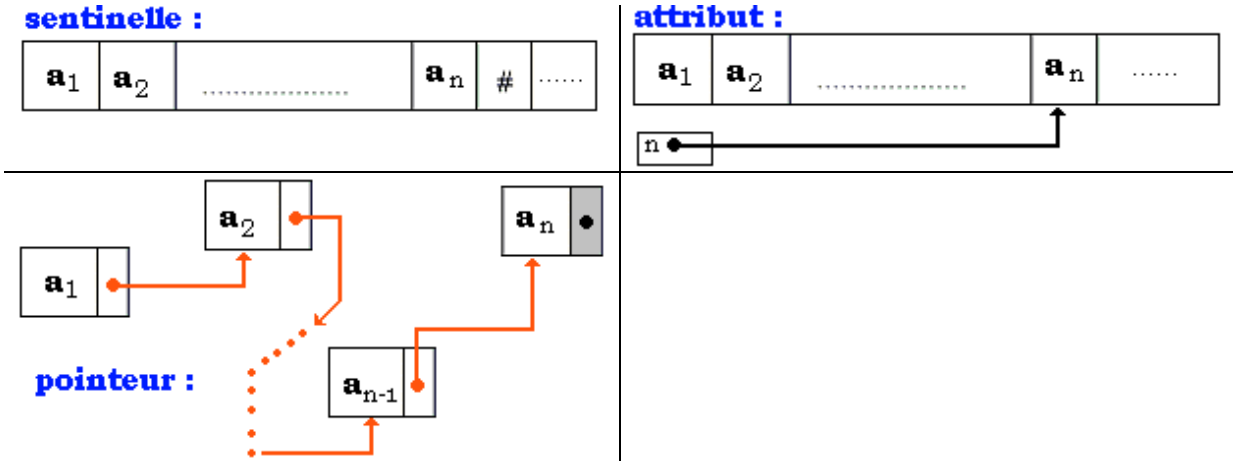
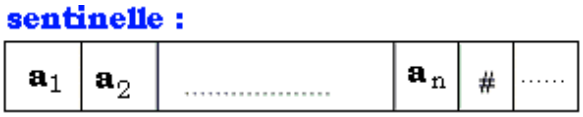


Fig - schéma des trois représentations des données (a_1, … , a_n)

Nous engageons le lecteur à écrire à chaque fois l'algorithme associé et à le traduire en un programme pascal.

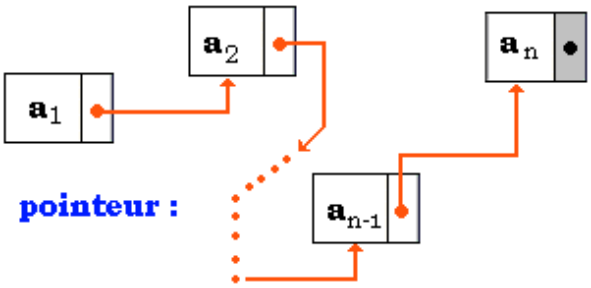
Nous donnons ci-après, au lecteur les trois versions d'implantation en pascal de la fonction CMP associée (sentinelle, pointeur, attribut).

CMP
programme avec sentinelle



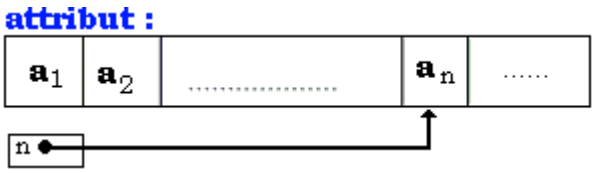

```
function CMP(ch1,ch2:Nom):Boolean;
var
   k : integer;
begin
  k:=1
  While (ch1[k]=ch2[k])
            and(ch1[k]<'#')
            and(ch2[k]<'#') do
     k:=k+1;
  if (ch1[k]='#')and(ch2[k]='#') then
        result :=True;
  if (ch1[k]='#')or(ch1[k]<ch2[k]) then
        result :=True;
  if (ch2[k]='#')or(ch2[k]<ch1[k]) then
        result :=False;
end;{CMP}
```

CMP
programme avec pointeur




```
type pchaine=^chaine;
 chaine=record
   car:char;
   suiv:pchaine;
 end;


function CMP(ch1,ch2:Nom):Boolean;
begin
 while ((ch1^.car=ch2^.car)
    and (ch1^.suiv<nil)
    and (ch2^.suiv<nil))      do
 begin
  ch1:=ch1^.suiv;
  ch2:=ch2^.suiv;
 end;
 if ((ch1^.suiv=nil) and (ch2^.suiv=nil)) then
CMP:=true;
 if (((ch1^.suiv=nil) and (ch2^.suiv<nil))
    or (ch1^.car<ch2^.car)) then result:=true;
 if (((ch2^.suiv=nil) and (ch1^.suiv<nil))
  or (ch1^.carch2^.car)) then result:=false;
end;{CMP}
```

CMP
programme avec attribut




```
const MaxCar=1000;
type inter=0..MaxCar;
chaine=record
  long:integer;
  car:array[1..MaxCar] of char;
end;


function CMP(ch1,ch2:Nom):Boolean;
var n:integer;
begin
 n:=1;
 while (ch1.car[n]=ch2.car[n])
          and ((n<n1)
          and (n<n2)) do
      n:=n+1;
 if ((n=ch1.long) and (n=ch2.long)) then
      result :=true;
 if (((n=ch1.long) and (n<ch2.long))
          or (ch1.car[n]<ch2.car[n])) then
     result:=true;
 if((n=ch2.long) and (n<ch1.long))
          or (ch1.car[n]ch2.car[n]) then
     result:=false;
end;{CMP}
```

3.2 : Modularité

Plan du chapitre:

1. La modularité

1.1 Notion de module

1.2 Critères principaux de modularité
- **La décomposabilité modulaire**
- **La composition modulaire**
- **La continuité modulaire**
- **La compréhension modulaire**
- **La protection modulaire**

1.3 Préceptes minimaux de construction modulaire
- **Interface de données minimale**
- **Couplage minimal**
- **Interfaces explicites**
- **Information publique et privée**

2. La modularité par les unit en pascal UCSD

2.1 Partie " public " d'une UNIT : " Interface "
2.2 Partie " privée " d'une UNIT : " Implementation "
2.3 Partie initialisation d'une UNIT

1. Modularité (selon B.Meyer)

1.1 Notion de module

Le mot **MODULE** est un des mots les plus employés en programmation moderne. Nous allons expliquer ici ce que l'on demande, à une méthode de construction modulaire de logiciels, de posséder comme propriétés, puis nous verrons comment dans certaines extensions de Pascal sur micro-ordinateur (Pascal, Delphi), cette notion de module se trouve implantée.

B.Meyer est l'un des principaux auteurs avec G.Booch qui ont le plus travaillé sur cette notion. Le premier a implanté ses idées dans le langage orienté objet " Eiffel ", le second a utilisé la modularité du langage Ada pour introduire le concept d'objet qui a été la base de méthodes de conception orientées objet : OOD, HOOD,UML...

Nous nous appuyons ici sur les concepts énoncés par B.Meyer fondés sur 5 critères et 6 principes relativement à une méthodologie d'analyse de type modulaire. Une démarche (et donc le logiciel construit qui en découle) est dite modulaire si elle respecte au moins les concepts ci-après.

1.2 Critères principaux de modularité

Les 5 principes retenus :

- **La décomposabilité modulaire**
- **La composition modulaire**
- **La continuité modulaire**
- **La compréhension modulaire**
- **La protection modulaire**

Définitions et réalisations en Pascal de ces cinq principes

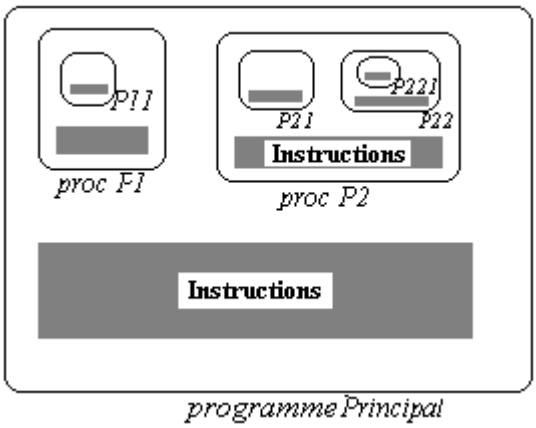
La décomposabilité modulaire :

capacité de décomposer un problème en sous-problèmes, semblable à la méthode structurée descendante.

***Réalisation de ce critère en Pascal* :**

La hiérarchie descendante des procédures et des fonctions.

Illustration de la décomposabilité en Pascal :



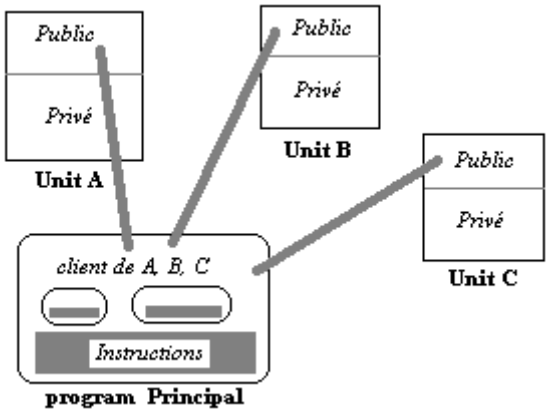
La composition modulaire :

capacité de recombinaison et de réagencement de modules écrits, semblable à la partie ascendante de la programmation structurée.

Réalisation de ce critère en Pascal :

N'existe pas en Pascal standard, toutefois la notion d'**UNIT** en Pascal UCSD (dont le Delphi est un descendant) et de **Library** en sont deux implantations partielles.

Illustration de la composition en Pascal :



La continuité modulaire :

capacité à réduire l'impact de changements dans les spécifications à un minimum de modules liés entre eux, et mieux à un seul module.

Réalisation de ce critère en Pascal :

Partiellement ; le cas particulier des constantes symboliques en Pascal standard, au paragraphe **const,** montre l'intérêt de ce critère**.**

Exemple : **const** n=10 ;
....
for i :=1 **to** n **do** ....
...
if T1 < n **then** ....

Il suffit de changer la ligne const n=10 pour modifier automatiquement les instructions où intervient la constante n, sans avoir à les réécrire toutes. Cette pratique en Pascal est très utile en particulier lorsqu'il s'agit de compenser le défaut dans la continuité modulaire, apporté par la notion de tableau statique dont les bornes doivent être connues à l'avance.

La compréhension modulaire :

capacité à l'interprétation par un programmeur du fonctionnement d'un module ou d'un ensemble de modules liés, sans avoir à connaître tout le logiciel.

Réalisation de ce critère en Pascal :

Partiellement à la charge du programmeur en écrivant des procédures et des fonctions qui s'appellent le moins possible. Chaque procédure ou fonction doit être dédiée à une tâche autonome.

Plus efficace dans Delphi grâce à la notion d'UNIT.

La protection modulaire :

capacité à limiter les effets produits par des incidents lors de l'exécution à un nombre minimal de modules liés entre eux, mieux à un seul module.

Réalisation de ce critère en Pascal :

Correcte en Pascal grâce au contrôle des types et des bornes des paramètres d'entrées ou de sorties d'une procédure ou d'une fonction. Les variables locales permettent de restreindre la portée d'un incident.

Attention *Les pointeurs en Pascal*

Le type pointeur met fortement en défaut ce critère, car sa gestion mémoire est de bas niveau et donc confiée au programmeur ; les pointeurs ne respectent même pas la notion de variable locale!
En général le passage par adresse met en défaut le principe de protection modulaire.

1.3 Préceptes minimaux de construction modulaire

Etant débutants, nous utiliserons quatre des six préceptes énoncés par B.Meyer. Ils sont essentiels et sont adoptés par tous ceux qui pratiquent des méthodes de programmation modulaire :

- **Interface de données minimale**
- **Couplage minimal**
- **Interfaces explicites**
- **Information publique et privée**

Précepte 1 : Interface de données minimale

Un module fixé doit ne communiquer qu'avec un nombre " minimum " d'autres modules du logiciel. L'objectif est de minimiser le ***nombre*** d'interconnexions entre les modules. Le graphe établissant les liaisons entre les modules est noté " graphe de dépendance ". Il doit être le moins maillé possible. La situation est semblable à celle que nous avons rencontré lors de la description des différentes topologies des réseaux d'ordinateurs : les liaisons les plus simples sont les liaisons en étoile, les plus complexes (donc ici déconseillées) sont les liaisons totalement maillées.

L'intérêt de ce précepte est de garantir un meilleur respect des critères de continuité et de protection modulaire. Les effets d'une modification du code source ou d'une erreur durant l'exécution dans un module peuvent se propager à un nombre plus ou moins important de modules en suivant le graphe de liaison. Un débutant optera pour une architecture de liaison simple, ce qui induira une construction contraignante du logiciel. L'optimum est défini par le programmeur avec l'habitude de la programmation.

Réalisation de ce précept en Pascal **:**

Le graphe de dépendance des procédures et des fonctions sera arborescent ou en étoile.

Précepte 2 : Couplage minimal

Lorsque deux modules communiquent entre eux, l'échange d'information doit être minimal. Ce précepte ne fait pas double emploi avec le précédent. Il s'agit de minimiser la *taille* des interconnexions entre modules et non leur *nombre* comme dans le précepte précédent.

Réalisation de ce précept en Pascal **:**

En général, nous avons aussi un couplage fort lorsqu'on introduit **toutes** les variables comme globales (donc à éviter, ce qui se produit au stade du débutant). D'autre part la notion de visibilité dans les blocs imbriqués et la portée des variables Pascal donne accès à des données qui ne sont pas toutes utiles au niveau le plus bas.

Précepte 3 : Interfaces explicites

Lorsque deux modules M1 et M2 communiquent, l'échange d'information doit être lisible explicitement dans l'un des deux ou dans les deux modules.

Réalisation de ce précept en Pascal **:**

L'utilisation des données globales ou de la notion de visibilité nuit aussi à ce principe. Le risque de battre en brèche le précepte des interfaces explicites est alors de conduire à des accès de données injustifiés (problème classique des effets de bord, où l'on utilise implicitement dans un bloc une donnée visible mais non déclarée dans ce bloc).

Précepte 4 : Information publique et privée

Toute information dans un module doit être répartie en deux catégories : l'information privée et l'information publique.

Ce précepte permet de modifier la partie privée sans que les clients (*modules utilisant ce module*) aient à supporter un quelconque problème à cause de modifications ou de

changements. Plus la partie publique est petite, plus on a de chances que des changements n'aient que peu d'effet sur les clients du module.

- La partie publique doit être la description des opérations ou du fonctionnement du module.
- La partie privée contient l'implantation des opérateurs et tout ce qui s'y rattache.

Réalisation de ce précept en Pascal :

Le Pascal standard ne permet absolument pas de respecter ce principe dans le cadre général. Delphi, avec la notion d'UNIT cotient une approche partielle mais utile de ce principe. Les prémisses de cette approche existent malgré tout dans les notions de variables et de procédures locales à une procédure. La notion de classe en Delphi implante complètement ce principe.

Enfin et pour mémoire nous citerons l'existence du précepte d'ouverture-fermeture et du précepte d'unités linguistiques.

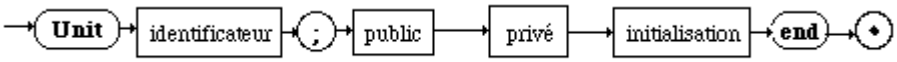
2. La modularité par les Unit avec Delphi

La notion de UNIT a été introduite en Pascal UCSD ancêtre de Delphi

Rappelons que Delphi et les versions de compilateurs libres gratuit comme FreePascal compiler, Obéron etc... présentes sur Internet fonctionnant sur les micro-ordinateurs type PC, ainsi que le Think Pascal de Symantec fonctionnant sur les MacIntosh d'Apple, sont tous une extension du Pascal UCSD. Il est donc possible sur du matériel courant d'utiliser la notion d'UNIT simulant le premier niveau du concept de module.

Cet élément représente une unité compilable séparément de tout programme et stockable en bibliothèque. Une **Unit** comporte une partie " public " et une partie " privé ". Elle implante donc l'idée de module et étend la notion de bloc (procédure ou fonction) en Pascal.

Syntaxe :



Exemple :

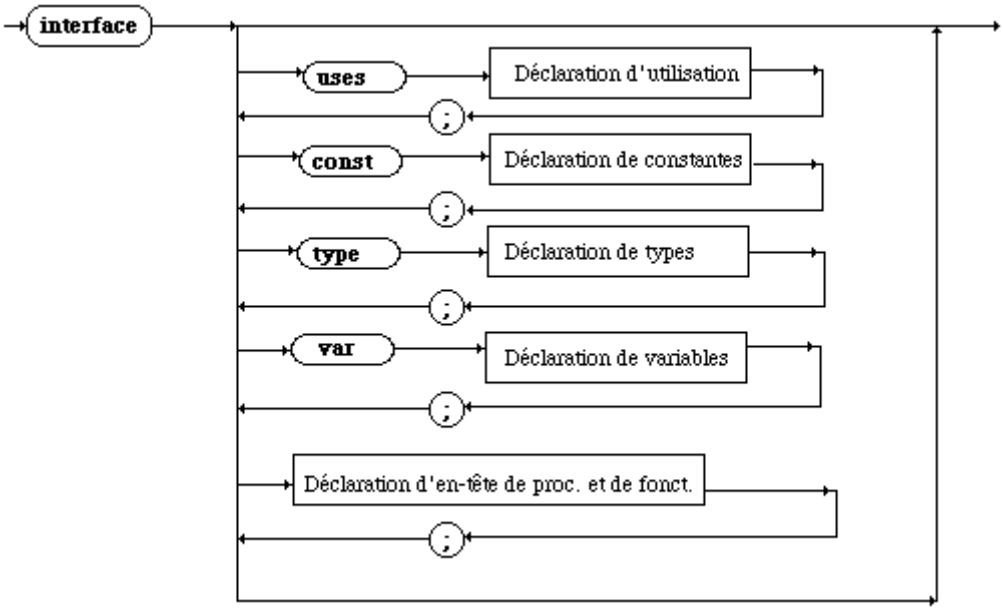
```
Unit Truc;
<partie public>
<partie privée>
<initialisation>
end.
```

2.1 Partie " public " d'une UNIT : " Interface "

Correspond exactement à la partie publique du module représenté par la UNIT. Cette partie décrit les en-têtes des procédures et des fonctions publiques utilisables par les clients. Les clients peuvent être soit d'autres procédures Pascal, des programmes Delphi ou d'autres Unit.

La clause Uses XXX dans un programme Delphi, permet d'indiquer la référence à la Unit XXX et autorise l'accès aux procédures et fonctions publiques de l'interface dans tout le programme.

Syntaxe :



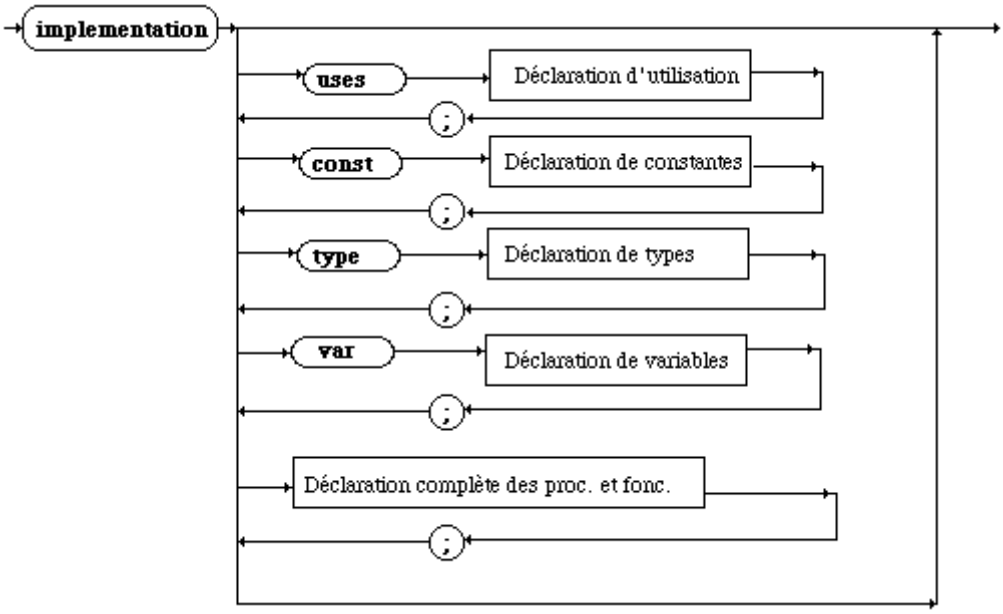
Exemple :

```
Unit Truc ;
interface
Uses Machin, Chose;
const
   a=10;
   x='a';
Type
   amoi=12..36;
var
   x, y : integer;
    z : amoi;
implementation

end.
```

2.2 Partie " privée " d'une UNIT : " Implementation "

Correspond à la partie privée du module représenté par la UNIT. Cette partie intimement liée à l'interface, contient le code interne du module. Elle contient deux sortes d'éléments : les déclarations complètes des procédures et des fonctions privées ainsi que les structures de données privées. Elle contient aussi les déclarations complètes des fonctions et procédures publiques dont les en-têtes sont présentes dans l'interface.

Syntaxe :



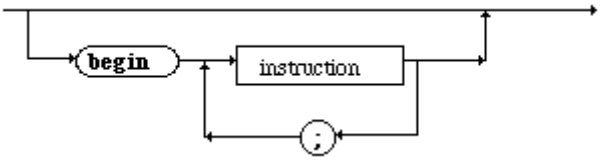
Exemple :

```
Unit Truc ;
interface
Uses Machin, Chose;
const
   a=10;
   x='a';

Type
   amoi = 12..36;

var
  x, y : integer;
  z : amoi;


procedure P1(x:real;var u:integer);
procedure P2(u,v:char;var x,y,t:amoi);
function F(x:real):boolean;
```

```
implementation

procedure P1(x:real;var u:integer);
begin
< corps de procédure>
end;

procedure P2(u,v:char;var x,y,t:amoi);
begin
< corps de procédure>
end;

function F(x:real):boolean;
begin
< corps de fonction>
end;

end.
```

2.3 Partie initialisation d'une UNIT

Il est possible d'initialiser des variables et d'exécuter des instructions au lancement de l'UNIT. Elles correspondent à des instructions classiques Pascal sur des données publiques ou privées de la **Unit** (initialisation de tableaux, mise à zéro de divers indicateurs, chargement de fichiers etc...):



3.3 : Complexité, tri, recherche

Plan du chapitre:

1. Complexité d'un algorithme

1.1 Notions de complexité temporelle et spatiale
1.2 Mesure de la complexité temporelle d'un algorithme
1.3 Notation de Landau O(n)

2. Trier des tableaux en mémoire centrale

2.1 Tri interne, tri externe
2.2 Des algorithmes classiques de tri interne

- *Le Tri à bulles*
- *Le Tri par sélection*
- *Le ri par insertion*
- *Le Tri rapide QuickSort*
- *Le Tri par tas HeapSort*

3. Rechercher dans un tableau

3.1 Recherche dans un tableau non trié
3.2 Recherche dans un tableau trié

1. Complexité d'un algorithme et performance

Nous faisons la distinction entre les méthodes (algorithmes) de tri d'un grand nombre d'éléments (plusieurs milliers ou plus), et le tri de quelques éléments (quelques dizaines, voir quelques centaines). Pour de très petits nombres d'éléments, la méthode importe peu. Il est intéressant de pouvoir comparer différents algorithmes de tris afin de savoir quand les utiliser. Ce que nous énonçons dans ce paragraphe s'applique en général à tous les algorithmes et en particulier aux algorithmes de tris qui en sont une excellente illustration.

1.1 Notions de complexité temporelle et spatiale

L'efficacité d'un algorithme est directement liée au programme à implémenter sur un ordinateur. Le programme va s'exécuter en un **temps fini** et va mobiliser des **ressources mémoires** pendant son exécution; ces deux paramètres se dénomment **complexité temporelle** et **complexité spatiale**.

Dès le début de l'informatique les deux paramètres "temps d'exécution" et "place mémoire" ont eu une importance à peu près égale pour comparer l'efficacité relative des algorithmes. Il est clair que depuis que l'on peut, à coût très réduit avoir des mémoires centrales d'environ 1 Giga octets dans une machine personnelle, les soucis de place en mémoire centrale qui s'étaient fait jour lorsque l'on travaillait avec des mémoires centrales de 128 Kilo octets (pour des gros matériels de recherche des années 70) sont repoussés psychologiquement plus loin pour un utilisateur normal. Comme c'est le système d'exploitation qui gère la mémoire disponible (RAM, cache, virtuelle etc...), les analyses de performances de gestion de la mémoire peuvent varier pour le même programme.

Le facteur temps d'exécution reste l'élément qualitatif le plus perceptible par l'utilisateur d'un programme ne serait ce que parce qu'il attend derrière son écran le résultat d'un travail qui représente l'exécution d'un algorithme.

L'informatique reste une science de l'ingénieur ce qui signifie ici, que malgré toutes les études ou les critères théoriques permettant de comparer l'efficacité de deux algorithmes dans l'absolu, dans la pratique nous ne pourrons pas dire qu'il y a un **meilleur** algorithme pour résoudre tel type de problème. Une méthode pouvant être lente pour certaines configurations de données et dans une autre application qui travaille systématiquement sur une configuration de données favorables la méthode peut s'avérer être la "meilleure".
La recherche de la performance à tout prix est aussi inefficace que l'attitude contraire.

Prenons à notre compte les recommandations de R.Sedgewick :

> Quel que soit le problème mettez d'abord en œuvre l'algorithme le plus simple, solution du problème, car le temps nécessaire à l'implantation et à la mise au point d'un algorithme "optimisé" peut être bien plus important que le temps requis pour simplement faire fonctionner un programme légèrement moins rapide.

Il nous faut donc un outil permettant de comparer l'efficacité ou complexité d'un algorithme à celle d'un autre algorithme résolvant le même problème.

1.2 Mesure de la complexité temporelle d'un algorithme

- 1.2.1 La complexité temporelle
- 1.2.2 Complexité d'une séquence d'instructions
- 1.2.3 Complexité d'une instruction conditionnelle
- 1.2.4 Complexité d'une itération finie bornée

Nous prenons le parti de nous intéresser uniquement au temps théorique d'exécution d'un algorithme. Pourquoi théorique et non pratique ? Parce que le temps pratique d'exécution d'un programme, comme nous l'avons signalé plus haut dépend :

- de la machine (par exemple processeur travaillant avec des jeux d'instructions optimisées ou non),
- du système d'exploitation (par exemple dans la gestion multi-tâche des processus),
- du compilateur du langage dans lequel l'algorithme sera traduit (compilateur natif pour un processeur donné ou non),
- des données utilisées par le programme (nature et/ou taille),
- d'un facteur intrinsèque à l'algorithme.

Nous souhaitons donc pouvoir utiliser un instrument mathématique de mesure qui rende compte de l'efficacité spécifique d'un algorithme indépendamment de son implantation en langage évolué sur une machine. Tout en sachant bien que certains algorithmes ne pourront pas être analysés ainsi soit parce que mathématiquement cela est impossible, soit parce que les configurations de données ne sont pas spécifiées d'un manière précise, soit parce que le temps mis à analyser correctement l'algorithme dépasserait le temps de loisir et de travail disponible du développeur !

Notre instrument, la complexité temporelle, est fondé sur des outils abstraits (qui ont leur correspondance concrète dans un langage de programmation). L'outil le plus connu est l'opération élémentaire (quantité abstraite définie intuitivement ou d'une manière évidente par le développeur).

Notion d'opération élémentaire

Une opération élémentaire est une opération fondamentale d'un algorithme si le temps d'exécution est directement lié (par une formule mathématique ou empirique) au nombre de ces opérations élémentaires. Il peut y avoir plusieurs opérations élémentaires dans un même algorithme.

Nous pourrons ainsi comparer deux algorithmes résolvant le même problème en comparant ce nombre d'opérations élémentaires effectuées par chacun des deux algorithmes.

1.2.1 La complexité temporelle : notation

C'est le décompte du nombre d'opérations élémentaires effectuées par un algorithme donné.

Il n'existe pas de méthodologie systématique (art de l'ingénieur) permettant pour un algorithme quelconque de compter les opérations élémentaires. Toutefois des règles usuelles sont communément admises par la communauté des informaticiens qui les utilisent pour évaluer la complexité temporelle.

Soient $\mathbf{i_1}$ **,** $\mathbf{i_2}$ **, ... ,** $\mathbf{i_k}$ des instructions algorithmiques (affectation, itération, condition,...)
Soit une opération élémentaire dénotée **OpElem**, supposons qu'elle apparaisse n_1 fois dans l'instruction $\mathbf{i_1}$, n_2 fois dans l'instruction $\mathbf{i_2}$, ... n_k fois dans l'instruction $\mathbf{i_k}$**.** Nous noterons $Nb(i_1)$ le nombre n_1, $Nb(i_2)$ le nombre n_2 etc.

Nous définissons ainsi la fonction Nb (i_k) indiquant le nombre d'opérations élémentaires dénoté **OpElem** contenu dans l'instruction algorithmique i_k :

Nb( ) : Instruction → Entier .

1.2.2 Complexité temporelle d'une séquence d'instructions

Soit **S** la séquence d'exécution des instructions $\mathbf{i_1}$ **;** $\mathbf{i_2}$ **; ... ;** $\mathbf{i_k}$, soit n_k =Nb (i_k) le nombre d'opérations élémentaires de l'instruction $\mathbf{i_k}$ **.**

Le nombre d'opérations élémentaires **OpElem** de S, Nb(**S**) est égal par définition à la somme des nombres: $n_1 + n_2 + ... + n_k$:

S =
début
$\mathbf{i_1}$ **;**
$\mathbf{i_2}$ **;**
... ;
$\mathbf{i_k}$
fin

$$Nb(\mathbf{S}) = \sum Nb(i_p) = n_1 + n_2 + ... + n_k$$

1.2.3 Complexité temporelle d'une instruction conditionnelle

Dans les instructions conditionnelles étant donné qu'il n'est pas possible d'une manière générale de déterminer systématiquement quelle partie de l'instruction est exécutée (le **alors** ou le **sinon**), on prend donc un majorant :

Cond =
si Expr **alors** E1
sinon E2
fsi

$$Nb(\mathbf{Cond}) < Nb(Expr) + \max(Nb(E1), Nb(E2))$$

1.2.4 Complexité temporelle d'une itération finie bornée

Dans le cas d'une boucle finie bornée (comme pour...fpour) contrôlée par une variable d'indice "i", l'on connaît le nombre exact d'itérations noté ***Nbr_d'itérations*** de l'ensemble des instructions composant le corps de la boucle dénotées S (où S est une séquence d'instructions), l'arrêt étant assuré par la condition de sortie Expr(i) dépendant de la variable d'indice de boucle i.

La complexité est égale au produit du nombre d'itérations par la somme de la complexité de la séquence d'instructions du corps et de celle de l'évaluation de la condition d'arrêt Expr(i).

Iter =

Itération Expr(i)
  S
finItér

Nb(**Iter**) = [Nb(S) + Nb(Expr(i))] x ***Nbr_d'itérations***

*Exemple dans le cas d'une boucle **pour**...**fpour** :*

Iter =

pour i<-- a **jusquà** b **faire**
  $\mathbf{i}_1$ **;**
  $\mathbf{i}_2$ **;**
  ... ;
  $\mathbf{i}_k$
fpour

La complexité de la condition d'arrêt est par définition de **1** (<= le temps d'exécution de l'opération effectuée en l'occurence un test, ne dépend ni de la taille des données ni de leurs valeurs), en **notant |b-a| le nombre exact d'itérations exécutées** (lorsque les bornes sont des entiers **|b-a|** vaut exactement la valeur absolue de la différence des bornes) nous avons :
Nb(**Iter**) = ($\sum$ Nb($\mathbf{i}_p$) + 1) . |b-a|

Lorsque le nombre d'itérations n'est pas connu mais seulement majoré (nombre noté ***Majorant_Nbr_d'itérations***), alors on obtient un majorant de la complexité de la boucle (le majorant correspond à la complexité dans le pire des cas).

Complexité temporelle au pire :

Majorant_Nb(**Iter**) = [Nb(S) + Nb(Expr(i))] x ***Majorant_Nbr_d'itérations***

1.3 Notation de Landau O(n)

Nous avons admis l'hypothèse qu'en règle générale **la complexité en temps** dépend de la taille **n** des données (plus le nombre de données est grand plus le temps d'exécution est long).

Cette remarque est d'autant plus proche de la réalité que nous étudierons essentiellement des algorithmes de tri dans lesquels les **n** données sont représentées par une liste à **n** éléments.

Afin que notre instrument de mesure et de comparaison d'algorithmes ne dépende pas de la machine physique, nous n'exprimons pas le temps d'exécution en unités de temps (millisecondes etc..) mais en unité de taille des données.

Nous ne souhaitons pas ici rentrer dans le détail mathématique des notations O(f(**n**)) de Landau sur les infiniment grands équivalents, nous donnons seulement une utilisation pratique de cette notation.

Pour une fonction f (n) dépendant de la variable n, on écrit :
f est O(g(n)) g(n) où g est elle-même une fonction de la variable entière n, et l'on lit **f est de l'ordre de grandeur de g(n)** ou plus succinctement **f est d'ordre g(n)**, lorsque :

f est d'ordre g(n) :

Pour tout valeur entière de **n**, il existe deux constantes a et b positives telles que : $a.g(\mathbf{n}) < f(\mathbf{n}) < b.g(\mathbf{n})$

Ce qui signifie que lorsque **n** tend vers l'infini (**n** devient très grand en informatique) le rapport f(**n**)/g(**n**) reste borné.

f est d'ordre g(n) :

$a < f(\mathbf{n})/g(\mathbf{n}) < b$ quand $\mathbf{n} \rightarrow \infty$
Lorsque **n** tend vers l'infini, le rapport f(**n**)/g(**n**) reste borné.

Exemple :

Supposons que f et g soient les polynômes suivants :

$f(n) = 3\mathbf{n}^2 - 7\mathbf{n} + 4$

$g(n) = \mathbf{n}^2$;

Lorsque **n** tend vers l'infini le rapport f(**n**)/g(**n**) tend vers 3:

$f(\mathbf{n})/g(\mathbf{n}) \rightarrow 3$ quand $\mathbf{n} \rightarrow \infty$

ce rapport est donc borné.

donc **f est d'ordre $\mathbf{n}^2$** ou encore **f est $O(\mathbf{n}^2)$**

C'est cette notation que nous utiliserons pour mesurer la complexité temporelle C en nombre d'opérations élémentaires d'un algorithme fixé. Il suffit pour pouvoir comparer des complexités temporelles différentes, d'utiliser les mêmes fonctions g(n) de base.

Les informaticiens ont répertorié des situations courantes et ont calculé l'ordre de complexité associé à ce genre de situation.

Les fonctions g(n) classiquement et pratiquement les plus utilisées sont les suivantes :

$g(n) = 1$
$g(n) = \log_k (n)$
$g(n) = n$
$g(n) = n.\log_k(n)$
$g(n) = n^2$

Ordre de complexité C	Cas d'utilisation courant
$g(n) = 1 \Rightarrow$ **C est O(1)**	Algorithme ne dépendant pas des données
$g(n) = \log_k (n) \Rightarrow$ **C est $O(\log_k(n))$**	Algorithme divisant le problème par une quantité constante (base k du logarithme)
$g(n) = n \Rightarrow$ **C est O(n)**	Algorithme travaillant directement sur chacune des n données
$g(n) = n.\log_k(n) \Rightarrow$ **C est $O(n.\log_k(n))$**	Algorithme divisant le problème en nombre de sous-problèmes constants (base k du logarithme), dont les résultats sont réutilisés par recombinaison
$g(n) = n^2 \Rightarrow$ **C est $O(n^2)$**	Algorithme traitant généralement des couples de données (dans deux boucles imbriquées).

2. Trier des tableaux en mémoire centrale

Un tri est une opération de classement d'éléments d'**une liste selon un ordre total** défini. Sur le plan pratique, on considère généralement deux domaines d'application des tris: les tris internes et les tris externes.

Que se passe-t-il dans un tri? On suppose qu'on se donne une suite de nombres entiers (ex: 5, 8, -3 ,6 ,42, 2,101, -8, 42, 6) et l'on souhaite les classer par ordre croissant (relation d'ordre au sens large). La suite précédente devient alors après le tri (classement) : (-8, -3, 2, 5, 6, 6, 8, 42, 42, 101). Il s'agit en fait d'une nouvelle suite obtenue par une permutation des éléments de la première liste de telle façon que les éléments résultants soient classés par ordre croissant au sens large selon la relation d'ordre totale " $\leq$ " : ($-8 \leq -3 \leq 2 \leq 5 \leq 6 \leq 6 \leq 8 \leq 42 \leq 42 \leq 101$).

Cette opération se retrouve très souvent en informatique dans beaucoup de structures de données. Par exemple, il faut établir le classement de certains élèves, mettre en ordre un dictionnaire, trier l'index d'un livre, etc...

2.1 Tri interne, tri externe

Un tri interne s'effectue sur des données stockées dans une table en mémoire centrale, **un tri externe** est relatif à une structure de données non contenue entièrement dans la mémoire centrale (comme un fichier sur disque par exemple).

Dans certains cas les données peuvent être stockées sur disque (mémoire secondaire) mais structurées de telle façon que chacune d'entre elles soit représentée en mémoire centrale par **une clef associée à un pointeur**. Le pointeur lié à la clef permet alors d'atteindre l'élément sur le disque (n° d'enregistrement...). Dans ce cas seules les clefs sont triées (en table ou en arbre) en mémoire centrale et il s'agit là d'un tri interne. Nous réservons le vocable tri externe uniquement aux manipulations de tris directement sur les données stockées en mémoire secondaire.

2.2 Des algorithmes classiques de tri interne

Dans les algorithmes référencés ci-dessous, nous notons ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) la liste à trier. Etant donné le mode d'accès en mémoire centrale (accès direct aux données) une telle liste est généralement implantée selon un tableau à une dimension de **n** éléments (cas le plus courant). Nous attachons dans les algorithmes présentés, à expliciter des tris majoritairement sur des tables, certains algorithmes utiliserons des structures d'arbres en mémoire centrale pour représenter notre liste à trier ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$).

Les opérations élémentaires principales les plus courantes permettant les calculs de complexité sur les tris, sont les suivantes :

Deux opérations élémentaires

La comparaison de deux éléments de la liste $\mathbf{a}_i$ et $\mathbf{a}_k$, (**si** $\mathbf{a}_i > \mathbf{a}_k$, **si** $\mathbf{a}i < \mathbf{a}k$,....) .
L'échange des places de deux éléments de la liste $\mathbf{a}_i$ et $\mathbf{a}_k$, (place ($\mathbf{a}_i$) $\leftrightarrow$ place ($\mathbf{a}_k$)).

Ces deux opérations seront utilisées afin de fournir une mesure de comparaison des tris entre eux. Nous proposons dans les pages suivantes cinq tris classiques, quatre concerne le tri de données dans un tableau, le cinquième est un tri de données situées dans un arbre binaire, ce dernier pourra en première lecture être ignoré, si le lecteur n'est pas familiarisé avec la notion d'arbre binaire

Tris sur des tables :

- Tri itératif à bulles
- Tri itératif par sélection
- Tri itératif par insertion
- Tri récursif rapide QuickSort

Tris sur un arbre binaire :

- Le Tri par tas / HeapSort

Le tri à bulle

- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme Delphi - Java
- **F)** Assistant visuel

C'est le moins performant de la catégorie des tris par échange ou sélection, mais comme c'est un algorithme simple, il est intéressant à utiliser pédagogiquement.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri à bulle est de parcourir la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en intervertissant toute paire d'éléments consécutifs ($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) non ordonnés.

Ainsi après le premier parcours, l'élément maximum se retrouve en $\mathbf{a}_n$. On suppose que l'ordre s'écrit de gauche à droite (à gauche le plus petit élément, à droite le plus grand élément).
On recommence l'opération avec la nouvelle sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_{n-1}$), et ainsi de suite jusqu'à épuisement de toutes les sous-suites (la dernière est un couple).

Le nom de tri à bulle vient donc de ce qu'à la fin de chaque itération interne, les plus grands nombres de chaque sous-suite se déplacent vers la droite successivement comme des bulles de la gauche vers la droite.

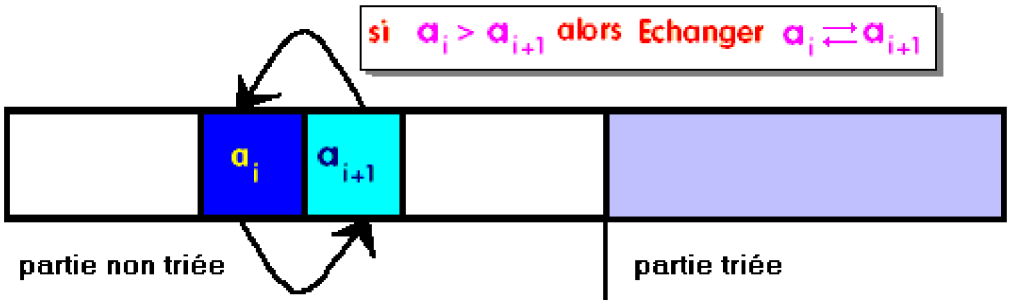
B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimension T[...] en mémoire centrale.

Le tableau contient une partie triée (en foncé à droite) et une partie non triée (en blanc à gauche).

On effectue plusieurs fois le parcours du tableau à trier.

Le principe de base est de ré-ordonner les couples ($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) non classés (en inversion de rang soit $\mathbf{a}_{i-1} > \mathbf{a}_i$) dans la partie non triée du tableau, puis à déplacer la frontière (le maximum de la sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_{n-1}$)) d'une position :



Tant que la partie non triée n'est pas vide, on permute les couples **non ordonnés** (($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) tels que $\mathbf{a}_{i-1} > \mathbf{a}_i$)) pour obtenir le maximum de celle-ci à l'élément frontière. C'est à dire qu'au premier passage c'est l'extremum global qui est bien classé, au second passage le second extremum etc...

C) Algorithme :

```
Algorithme Tri_a_Bulles
 local:   i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
début
 pour i de n jusquà 1 faire // recommence une sous-suite (a1, a2, ... , ai)
  pour j de 2 jusquà i faire // échange des couples non classés de la sous-suite
   si Tab[ j-1 ] > Tab[ j ] alors // aj-1 et aj non ordonnés
     temp ← Tab[ j-1 ] ;
     Tab[ j-1 ] ← Tab[ j ] ;
     Tab[ j ]  ← temp //on échange les positions de aj-1 et aj
   Fsi
  fpour
 fpour
Fin Tri_a_Bulles
```

Exemple : soit la liste (**5 , 4 , 2 , 3 , 7 , 1**), appliquons le tri à bulles sur cette liste d'entiers. Visualisons les différents états de la liste pour chaque itération externe contôlée par l'indice i :

i = 6 / pour j **de** 2 **jusquà** 6 **faire**

5	4	2	3	7	1	*5 > 4 donc permutation des deux cellules*
4	5	2	3	7	1	*5 > 2 donc permutation des deux cellules*
4	2	5	3	7	1	*5 > 3 donc permutation des deux cellules*
4	2	3	5	7	1	*5 < 7 donc aucune action sur ces deux cellules*
4	2	3	5	7	1	*7 > 1 donc permutation des deux cellules*
4	2	3	5	1	7	*A la fin de la boucle externe le max 7 est rangé*

i = 5 / pour j **de** 2 **jusquà** 5 **faire**

4	2	3	5	1	7	*4 > 2 donc permutation des deux cellules*
2	4	3	5	1	7	*4 > 3 donc permutation des deux cellules*
2	3	4	5	1	7	*4 < 5 donc aucune action sur ces deux cellules*
2	3	4	5	1	7	*5 > 1 donc permutation des deux cellules*
2	3	4	1	5	7	*A la fin de la boucle externe le max 5 est rangé*

i = 4 / pour j **de** 2 **jusquà** 4 **faire**

2	3	4	1	5	7	*2 < 3 donc aucune action sur ces deux cellules*
2	3	4	1	5	7	*3 < 4 donc aucune action sur ces deux cellules*
2	3	4	1	5	7	*4 > 1 donc permutation des deux cellules*
2	3	1	4	5	7	*A la fin de la boucle externe le max 4 est rangé*

i = 3 / pour j **de** 2 **jusquà** 3 **faire**

2	3	1	4	5	7	*2 < 3 donc aucune action sur ces deux cellules*
2	3	1	4	5	7	*3 > 1 donc permutation des deux cellules*
2	1	3	4	5	7	*A la fin de la boucle externe le max 3 est rangé*

i = 2 / pour j **de** 2 **jusquà** 2 **faire**

2	1	3	4	5	7	*2 > 1 donc permutation des deux cellules*
1	2	3	4	5	7	*A la fin de la boucle externe le max 2 est rangé*

i = 1 / pour j **de** 2 **jusquà** 1 **faire** ***(boucle vide)***

1	2	3	4	5	7	*Il ne reste plus d'éléments à comparer !*

D) Complexité :

Choix opération

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Le nombre de comparaisons "**si** Tab[j-1] > Tab[j] **alors**" est une valeur qui ne dépend que de la longueur **n** de la liste (**n** est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle "**pour** i **de** **n** **jusquà** 1 **faire**" s'exécute n fois (donc une somme de n termes) et qu'à chaque fois la boucle "**pour** j **de** 2 **jusquà** i **faire**" exécute (i-2)+1 fois la comparaison "**si** Tab[j-1] > Tab[j] **alors**".

La complexité en nombre de comparaison est égale à la somme des n termes suivants (i = n, i = n-1,.... i = 1)

C = (n-2)+1 + ([n-1]-2)+1 +.....+1+0 = (n-1)+(n-2)+...+1 = n(n-1)/2 (c'est la somme des n-1 premiers entiers).

La complexité du tri à bulle en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons comme opération élémentaire **l'échange de deux cellules** du tableau.

Calculons par dénombrement le nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échanges). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse et donc chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.

La complexité du tri à bulle au pire en nombre d'échanges est de l'ordre de n^2, que l'on écrit $O(n^2)$.

E) Programme Delphi (tableau d'entiers):

```
program TriParBulle;
const N = 10;  { Limite supérieure de tableau }
type TTab = array [1..N] of integer;   { TTab : Type Tableau }
var Tab : TTab ;

procedure TriBulle (var Tab:TTab) ;
{ Implantation Pascal de l'algorithme }
var i, j, t : integer;
begin
  for i := N downto 1 do
    for j := 2 to i do
      if Tab[j-1] > Tab[j] then
      begin
        t := Tab[j-1];
        Tab[j-1] := Tab[j];
        Tab[j] := t;
     end;
end;


procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
```

```
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
       writeln;
end;

begin
  Initialisation(Tab);
  writeln('TRI PAR BULLE');
  writeln;
  Impression(Tab);
  TriBulle(Tab);
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR BULLE

-------------------------------------------------------------
 32 |  60 |  60 |  54 |  70 |  53 |  64 |  91 |  69 |  81 |
-------------------------------------------------------------
 32 |  53 |  54 |  60 |  60 |  64 |  69 |  70 |  81 |  91 |
-------------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriBulle  {

 static int[] table = new int[10] ; // le tableau à trier en attribut

 static void TriBulle ( )  {
  int n = table.length-1;
  for ( int i = n; i>=1; i--)
   for ( int j = 2; j<=i; j++)
    if (table[j-1] > table[j])
    {
     int temp = table[j-1];
     table[j-1] = table[j];
     table[j] = temp;
    }
 }
```

```
 static void Impression ( ) {
  // Affichage du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void Initialisation  ( )  {
  // remplissage aléatoire du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
 }

 public static void main(String[ ] args)  {
  Initialisation ( );
  System.out.println("Tableau initial :");
  Impression ( );
  TriBulle ( );
  System.out.println("Tableau une fois trié :");
  Impression ( );
 }
}
```

Le tri par sélection

- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme Delphi - Java

C'est une version de base de la catégorie des tris par sélection.

A) Spécification abstraite

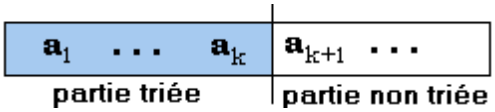
Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est décomposée en deux parties : une partie liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$) et une partie non-triée ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$); l'élément $\mathbf{a}_{k+1}$ est appelé élément frontière (c'est le premier élément non trié).

Le principe est de parcourir la partie non-triée de la liste (**a**k+1, **a**k+2, ... , **a**n) en cherchant l'élément minimum, puis en l'échangeant avec l'élément frontière **a**k+1, puis à déplacer la frontière d'une position. Il s'agit d'une récurrence sur les minima successifs. On suppose que l'ordre s'écrit de gauche à droite (à gauche le plus petit élément, à droite le plus grand élément).
On recommence l'opération avec la nouvelle sous-suite (**a**k+2, ... , **a**n) , et ainsi de suite jusqu'à ce que la dernière soit vide.

B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimensionT[...] en mémoire centrale.

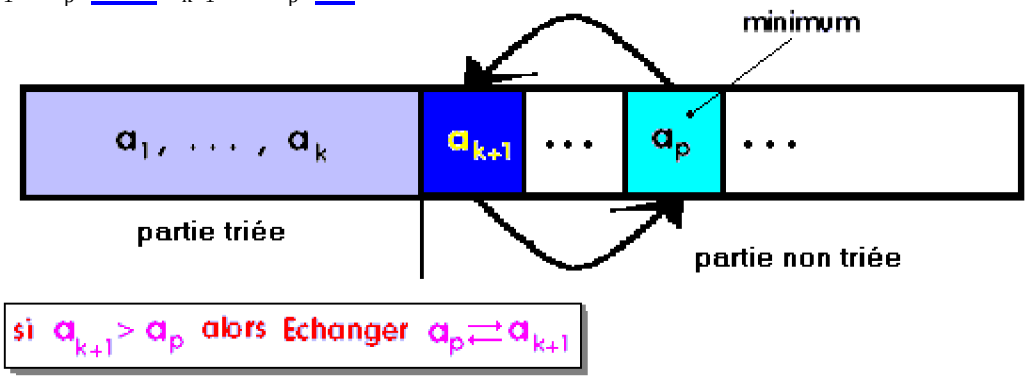
Le tableau contient une partie triée (en foncé à gauche) et une partie non triée (en blanc à droite).



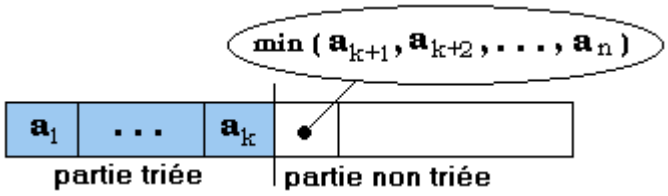
On cherche le minimum de la partie non-triée du tableau et on le recopie dans la cellule frontière (le premier élément de la partie non triée).

Donc pour tout $\mathbf{a}_p$ de la partie non triée on effectue l'action suivante :

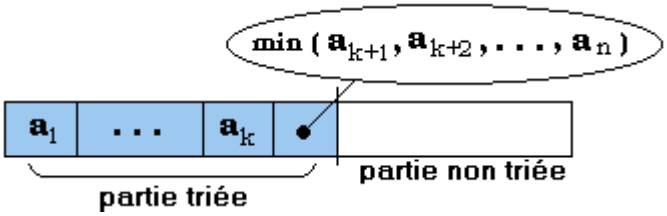
si $\mathbf{a}_{k+1} > \mathbf{a}_p$ **alors** $\mathbf{a}_{k+1} \leftarrow \mathbf{a}_p$ **Fsi**



et l'on obtient ainsi à la fin de l'examen de la sous-liste ($\mathbf{a}_{k+1}$, a_{k+2}, ... , $\mathbf{a}_n$) la valeur min ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$) stockée dans la cellule $\mathbf{a}_{k+1}$.



La sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$, $\mathbf{a}_{k+1}$) est maintenant triée :



Et l'on recommence la boucle de recherche du minimum sur la nouvelle sous-liste ($\mathbf{a}_{k+2}$, $\mathbf{a}_{k+3}$, ... , $\mathbf{a}_n$) etc...

Tant que la partie non triée n'est pas vide, on range le minimum de la partie non-triée dans l'élément frontière.

C) Algorithme :

Une version maladroite de l'algorithme mais exacte a été fournie par un groupe d'étudiants elle est dénommée / **Version maladroite 1**/.

Elle échange physiquement et systématiquement l'élément frontière Tab[i] avec un élément Tab[j] dont la valeur est plus petite (la suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_i$) est triée) :

Maladroit

Algorithme Tri_Selection /Version maladroite 1/
local: m, i , j , **n**, temp ∈ Entiers naturels
Entrée : Tab ∈ Tableau d'Entiers naturels de **1** à **n** éléments
Sortie : Tab ∈ Tableau d'Entiers naturels de **1** à **n** éléments

```
début
 pour i de 1 jusquà n-1faire // recommence une sous-suite
  m ← i ; // i est l'indice de l'élément frontière Tab[ i ]
  pour j de i+1 jusquà n faire// liste non-triée : (a_{i+1} , a_{i+2} , ... , a_n )
   si Tab[ j ] <  Tab[ m ] alors // a_j est le nouveau minimum partiel
    m ← j ;
    temp ←Tab[ m ] ;
    Tab[ m ] ← Tab[ i ] ;
    Tab[ i ]  ← temp //on échange les positions de a_i et de a_j
    m ← i ;
   Fsi
  fpour
 fpour
Fin Tri_Selection
```

Voici une version correcte et améliorée du précédent (nous allons voir avec la notion de complexité comment appuyer cette intuition d'amélioration), dans laquelle l'on sort l'échange a_i *et* a_j de la boucle interne "**pour** j **de** i+1 **jusquà n faire**" pour le déporter à la fin de cette boucle.

Amélioration

Au lieu de travailler sur les contenus des cellules de la table, nous travaillons sur les indices, ainsi lorsque ***aj*** est plus petit que ***ai*** nous mémorisons l'indice "j" du minimum dans une variable " m ← j ; " plutôt que le minimum lui-même.

Version maladroite	Version améliorée

pour j **de** i+1 **jusquà n faire**
  si Tab[j] < Tab[m] **alors**
  m ← j ;
  temp ←Tab[m] ;
  Tab[m] ← Tab[i] ;
  Tab[i] ← temp
  m ← i ;
  Fsi
fpour
	pour j **de** i+1 **jusquà n faire**
	  si Tab[j] < Tab[m] **alors**
	  m ← j ;
	  Fsi
	fpour;
	temp ← Tab[m] ;
	Tab[m] ←Tab[i] ;
	Tab[i] ← temp

A la fin de la boucle interne "**pour** j **de** i+1 **jusquà n faire**" la variable m contient l'indice de min(**a**i+1, **a**k+2, ... , **a**n) et l'on permute l'élément concerné (d'indice m) avec l'élément frontière ***ai*** :

```
Algorithme Tri_Selection  /Version 2 améliorée/
 local:   m, i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
        début
         pour i de 1 jusquà n-1 faire // recommence une sous-suite
          m ← i ; // i est l'indice de l'élément frontière ai = Tab[ i ]
          pour j de i+1 jusquà n faire// (a_{i+1} , a_{i+2} , ... , a_n )
           si Tab[ j ] <  Tab[ m ] alors // aj est le nouveau minimum partiel
             m ← j ; // indice mémorisé
           Fsi
          fpour;
          temp ← Tab[ m ] ;
          Tab[ m ] ←Tab[ i ] ;
          Tab[ i ]  ← temp //on échange les positions de a_i et de a_j
         fpour
        Fin Tri_Selection
```

D) Complexité :

Choix opération

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Pour les deux versions 1 et 2 :
Le nombre de comparaisons "**si** Tab[j] < Tab[m] **alors**" est une valeur qui ne dépend que de la longueur **n** de la liste (**n** est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle "**pour** i **de** 1 **jusquà n-1 faire**" s'exécute n-1 fois (donc une somme de n-1 termes) et qu'à chaque fois la boucle "**pour** j **de** i+1 **jusquà n faire**" exécute (n-(i+1)+1 fois la comparaison "**si** Tab[j] < Tab[m] **alors**".

La complexité en nombre de comparaison est égale à la somme des n-1 termes suivants (i = 1, ...i = n-1)
C = (n-2)+1 + (n-3)+1 +.....+1+0 = (n-1)+(n-2)+...+1 = n.(n-1)/2 (c'est la somme des n-1 premiers entiers).

La complexité du tri par sélection en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons comme opération élémentaire **l'échange de deux cellules** du tableau.

Calculons par dénombrement le nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échanges). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse.

Pour la version 1

Au pire chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.

La complexité au pire en nombre d'échanges de la version 1 est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Pour la version 2

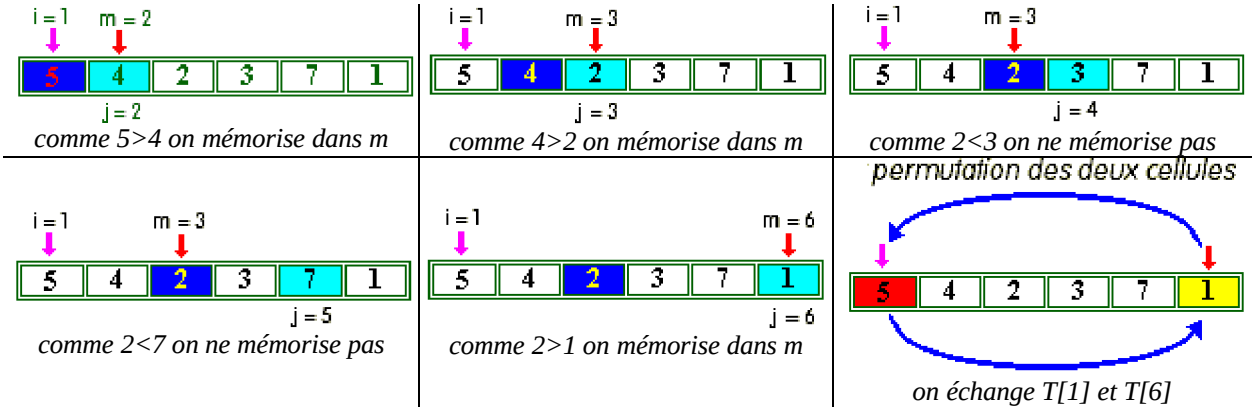
L'échange a lieu systématiquement dans la boucle principale "**pour** i **de** 1 **jusquà n-1 faire**" qui s'exécute n-1 fois :

La complexité en nombre d'échanges de cellules de la version 2 est de l'ordre de n, que l'on écrit O(n).

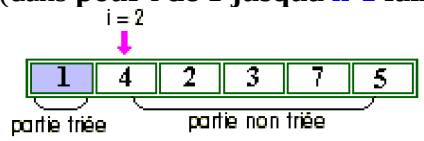
Un échange valant 3 transferts (affectation) la complexité en transfert est O(3n) = O(n)

Toutefois cette complexité en nombre d'échanges de cellules n'apparaît pas comme significative du tri, outre le nombre de comparaison, c'est le nombre d'affectations d'indice qui représente une opération fondamentale et là les deux versions ont exactement la même complexité **$O(n^2)$**.

Exemple : soit la liste à 6 éléments (**5 , 4 , 2 , 3 , 7 ,1**), appliquons la version 2 du tri par sélection sur cette liste d'entiers. Visualisons les différents états de la liste pour la première itération externe contrôlée par i (i = 1) et pour les itérations internes contrôlées par l'indice j (de j = 2 ... à ... j = 6) :



L'algorithme ayant terminé l'échange de T[1] et de T[6], il passe à l'itération externe suivante (dans **pour** i **de** 1 **jusquà** **n-1** **faire** , il passe à i = 2) :



etc....

E) Programme Delphi (tableau d'entiers) :

```
program TriParSelection;
const N = 10;  { Limite supérieure de tableau }
type TTab = array [1..N] of integer;   { TTab : Type Tableau }
var Tab : TTab ;

 procedure TriSelection (var Tab:TTab) ;
{ Implantation Pascal de l'algorithme }
var i, j, t, m : integer;
begin
  for i := 1 to N-1 do
  begin
    m := i;
    for j := i+1 to N do
      if Tab[ j ] < Tab[ m ] then m := j;
    t := Tab[m];
    Tab[m] := Tab[i];
    Tab[i] := t;
  end;
end;

procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
       writeln;
end;

begin
  Initialisation(Tab);
```

```
    writeln('TRI PAR SELECTION');
    writeln;
    Impression(Tab);
    TriSelection(Tab);
    Impression(Tab);
    writeln('----------------------');
  end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR SELECTION

-------------------------------------------------------------
 28 |  51 |  86 |  43 |  32 |   6 |  52 |  51 |  79 |  42 |
-------------------------------------------------------------
  6 |  28 |  32 |  42 |  43 |  51 |  51 |  52 |  79 |  86 |
-------------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriSelect
{
 static int[] table = new int[20] ; // le tableau à trier en attribut

 static void Impression ( )  {
  // Affichage du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void Initialisation  ( )  {
  // remplissage aléatoire du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
 }
```

```
 static void TriSelect ( ) {
  int n = table.length-1;
  for ( int i = 1; i <= n-1; i++)
  { // recommence une sous-suite
   int m = i; // élément frontière ai = table[ i ]
   for ( int j = i+1; j <= n; j++)   // (ai+1, a2, ... , an)
    if (table[ j ] < table[ m ]) // aj = nouveau minimum partiel
     m = j ; // indice mémorisé
   int temp = table[ m ];
   table[ m ] = table[ i ];
   table[ i ]= temp;
  }
 }
```

```
 public static void main(String[ ] args)
 {
  Initialisation ( );
  System.out.println("Tableau initial :");
  Impression ( );
  TriSelect ( );
  System.out.println("Tableau une fois trié :");
  Impression ( );
 }
}
```

Le tri par insertion

- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme Delphi - Java

C'est un tri en général un peu plus coûteux en particulier en nombre de transfert à effectuer qu'un tri par sélection (cf. complexité**).**

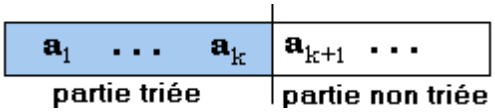
A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par insertion est de parcourir la liste non triée liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en la décomposant en deux parties : une partie déjà triée et une partie non triée.

La méthode est identique à celle que l'on utilise pour ranger des cartes que l'on tient dans sa main : on insère dans le paquet de cartes déjà rangées une nouvelle carte au bon endroit.

L'opération de base consiste à prendre l'élément frontière dans la partie non triée, puis à l'insérer à sa place dans la partie triée (place que l'on recherchera séquentiellement), puis à déplacer la frontière d'une position vers la droite. Ces insertions s'effectuent tant qu'il reste un élément à ranger dans la partie non triée.. L'insertion de l'élément frontière est effectuée par décalages successifs d'une cellule.

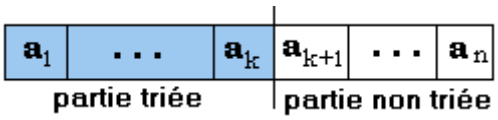
La liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est décomposée en deux parties : une partie triée ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$) et une partie non-triée ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$); l'élément $\mathbf{a}_{k+1}$ est appelé élément frontière (c'est le premier élément non trié).



B) Spécification concrète itérative

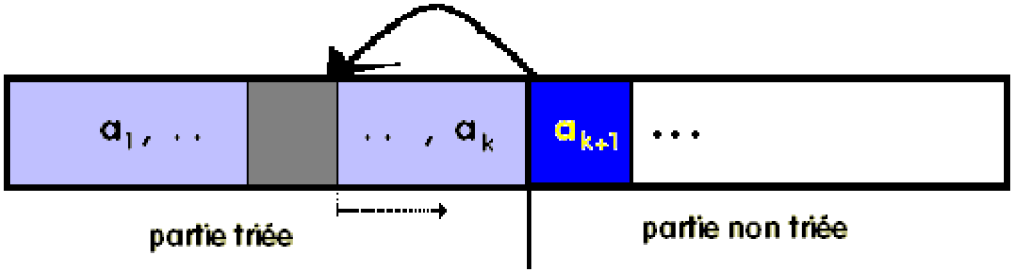
La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimension T[...] en mémoire centrale.

Le tableau contient une partie triée (($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$) en foncé à gauche) et une partie non triée (($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$); en blanc à droite) :



On insère l'élément frontière $\mathbf{a}_{k+1}$ en faisant varier j de k jusqu'à 2 , afin de balayer toute la partie (**a**1, **a**2, ... , **a**k) déjà rangée, on décale alors d'une place les éléments plus grands que l'élément frontière :

tantque $\mathbf{a}_{j-1} > \mathbf{a}_{k+1}$ **faire**
    décaler $\mathbf{a}_{j-1}$ en $\mathbf{a}_j$;
    passer au j précédent
ftant



La boucle s'arrête lorsque $\mathbf{a}_{j-1} < \mathbf{a}_{k+1}$,ce qui veut dire que l'on vient de trouver au rang j-1 un élément $\mathbf{a}_{j-1}$ plus petit que l'élément frontière $\mathbf{a}_{k+1}$, donc $\mathbf{a}_{k+1}$ doit être placé au rang j.

C) Algorithme :

Algorithme Tri_Insertion
local: i , j , **n**, v ∈ Entiers naturels
Entrée : Tab ∈ Tableau d'Entiers naturels de 0 à **n** éléments
Sortie : Tab ∈ Tableau d'Entiers naturels de 0 à **n** éléments
{ *dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la boucle* **tantque** .. **faire** *si l'indice j n'est pas inférieur à 1, elle aura une valeur inférieure à toute valeur possible de la liste*
}
début
pour i **de**2 **jusquà n faire**// *la partie non encore triée* (a_i , a_{i+1} , ... , a_n)
    v ←Tab[i] ; // *l'élément frontière :* a_i
    j ← i ; // *le rang de l'élément frontière*
    Tantque Tab[j-1] > v **faire** // *on travaille sur la partie déjà triée* (a_1 , a_2 , ... , a_i)
        Tab[j] ← Tab[j-1]; // *on décale l'élément*
        j ← j-1; // *on passe au rang précédent*

```
      FinTant ;
      Tab[ j ]  ← v //on recopie ai dans la place libérée
   fpour
 Fin Tri_Insertion
```

Sans la sentinelle en T[0] nous aurions une comparaison sur **j** à l'intérieur de la boucle :

```
Tantque Tab[ j-1 ] > v faire//on travaille sur la partie déjà triée(a1 , a2 , ... , ai)
          Tab[ j ]  ← Tab[ j-1 ]; // on décale l'élément
           j ← j-1;              // on passe au rang précédent
        si j = 0 alors Sortir de la boucle fsi
FinTant ;
```

Exercice

Un étudiant a proposé d'intégrer la comparaison dans le test de la boucle en écrivant ceci :

```
Tantque ( Tab[j-1] > v ) et ( j >0 ) faire
          Tab[ j ]  ← Tab[ j-1 ];
           j ← j-1;
FinTant ;
```

Il a eu des problèmes de dépassement d'indice de tableau lors de l'implémentation de son programme.

Essayez d'analyser l'origine du problème en notant que la présence d'une sentinelle élimine le problème.

D) Complexité :

Choix opération

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Dans le pire des cas le nombre de comparaisons "**Tantque** Tab[j-1] > v **faire**" est une valeur qui ne dépend que de la longueur **i** de la partie (**a**1, **a**2, ... , **a**i) déjà rangée. Il y a donc au pire i comparaisons pour chaque i variant de 2 à n :
La complexité au pire en nombre de comparaison est donc égale à la somme des n termes suivants (i = 2, i = 3,.... i = n)
C = 2 + 3 + 4 +...+ n = n(n+1)/2 -1 comparaisons au maximum. (c'est la somme des n premiers entiers moins 1).

La complexité au pire en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons maintenant comme opération élémentaire **le transfert d'une cellule** du tableau.

Calculons par dénombrement du nombre de transferts dans le pire des cas .

Il y a autant de transferts dans la boucle "**Tantque** Tab[j-1] > v **faire**" qu'il y a de comparaisons il faut ajouter 2 transferts par boucle "**pour** i **de** 2 **jusquà n faire**", soit au total dans le pire des cas :
$C = n(n+1)/2 + 2(n-1) = (n^2 + 5n - 4)/2$

La complexité du tri par insertion au pire en nombre de transferts est de l'ordre de n^2, que l'on écrit $O(n^2)$.

E) Programme Delphi (tableau d'entiers) :

```
program TriParInsertion;
const N = 10;  { Limite supérieure de tableau }
type TTab = array [0..N]of integer;   { TTab : Type Tableau }
var Tab : TTab ;

 procedure TriInsertion (var Tab:TTab) ;
{ Implantation Pascal de l'algorithme }
var i, j, v : integer;
begin
  for i := 2 to N do
  begin
     v := Tab[ i ];
     j := i ;
    while Tab[ j-1 ] > v do
    begin
        Tab[ j ] := Tab[ j-1 ] ;
         j := j - 1 ;
    end;
    Tab[ j ] := v ;
  end
end;

procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
  Tab[0]:= -Maxint ; // la sentinelle est l'entier le plus petit du type
integer sur la machine
end;
```

```
procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
      writeln;
end;

begin
  Initialisation(Tab);
  writeln('TRI PAR INSERTION');
  writeln;
  Impression(Tab);
  TriInsertion(Tab);
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR INSERTION

-------------------------------------------------------------
 62 |  15 |  34 |   3 |  25 |  22 |  63 |   3 |  66 |  17 |
-------------------------------------------------------------
  3 |   3 |  15 |  17 |  22 |  25 |  34 |  62 |  63 |  66 |
-------------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriInsert
{
 // le tableau à trier:
 static int[] table = new int[10] ;
 /*-----------------------------------------------------------------------
  Dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la boucle tantque .. faire si
  l'indice j n'est pas  inférieur à 1, elle aura une valeur inférieure à toute  valeur possible de la liste
  -----------------------------------------------------------------------*/
 static void Impression ( )  {
  // Affichage du tableau
  int n = table.length-1;
  for ( int i = 0; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void Initialisation  ( )  {
  // remplissage aléatoire du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
  //sentinelle à l'indice 0 :
  table[0] = -Integer.MAX_VALUE;
 }
```

```
public static void main(String[ ] args)  {
 Initialisation  ( );
 System.out.println("Tableau initial :");
 Impression ( );
 TriInsert ( );
 System.out.println("Tableau une fois trié :");
 Impression ( );
}
```

```
static void TriInsert ( ) {
 // sous-programme de Tri par insertion :
 int n = table.length-1;
 for ( int i = 2; i <= n; i++)
 {
  int v = table[i];
  int j = i;
  while (table[ j-1 ] > v)
  {// travail sur la partie déjà triée (a1, a2, ... , ai)
   table[ j ] = table[ j-1 ]; // on décale l'élément
   j--;  // on passe au rang précédent
  }
  table[ j ]  = v ; //on recopie ai dans la place libérée
 }
}
```

```
}
```

Le tri rapide

méthode Sedgewick

- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme Delphi - Java

C'est le plus performant des tris en table qui est certainement celui qui est le plus employé dans les programmes. Ce tri a été trouvé par C.A.Hoare, nous nous référons à Robert Sedgewick qui a travaillé dans les années 70 sur ce tri et l'a amélioré et nous renvoyons à son ouvrage pour une étude complète de ce tri. Nous donnons les principes de ce tri et sa complexité en moyenne et au pire.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par insertion est de parcourir la liste L = liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en la divisant systématiquement en deux sous-listes L1 et L2. L'une de ces deux sous-listes est telle que tous ses éléments sont inférieurs à tous ceux de l'autre liste , la division en sous-liste a lieu en travaillant séparément sur chacune des deux sous-listes en appliquant à nouveau la même division à chaque sous-liste jusqu'à obtenir uniquement des sous-listes à un seul élément.

C'est un algorithme dichotomique qui divise donc le problème en deux sous-problèmes dont les résultats sont réutilisés par recombinaison, il est donc de complexité **O(n.log(n))**.

Pour partitionner une liste L en deux sous-listes L1 et L2 :

- on choisit une valeur quelconque dans la liste L (la dernière par exemple) que l'on dénomme **pivot**,
- puis on construit la sous-liste L1 comme comprenant tous les éléments de L dont la valeur est inférieure ou égale au **pivot**,
- et l'on construit la sous-liste L2 comme constituée de tous les éléments dont la valeur est supérieure au **pivot**.

Soit sur un exemple de liste L :

L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]
prenons comme pivot la dernière valeur pivot = 16

Nous obtenons deux sous-listes L1 et L2 :

L1 = [4, 14, 3, 2]
L2 = [23, 45, 18, 38, 42]

A cette étape voici l'arrangement de L :

L = L1 + pivot + L2 = [**4, 14, 3, 2**, **16**, **23, 45, 18, 38, 42**]

En effet, en travaillant sur la table elle-même par réarrangement des valeurs, le pivot **16** est placé au bon endroit directement :

[**4<16, 14<16, 3<16, 2<16**, **16**, **23>16, 45>16, 18>16, 38>16, 42>16**]

En appliquant la même démarche au deux sous-listes : L1 (pivot=2) et L2 (pivot=42)
[**4, 14, 3, 2**, **16**, **23, 45, 18, 38, 42**] nous obtenons :

L11=[ ] liste vide
L12=[**3, 4, 14**]
L1=L11 + pivot + L12 = (**2,3, 4, 14**)

L21=[**23, 38, 18**]
L22=[**45**]
L2=L21 + pivot + L22 = (**23, 38, 18, 42, 45**)

A cette étape voici le nouvel arrangement de L :

L = [(**2,3, 4, 14**), **16**, (**23, 38, 18, 42, 45**)]

etc...
Ainsi
de proche en proche en subdivisant le problème en deux sous-problèmes, à chaque étape nous obtenons un pivot bien placé.

B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau de dimension unT[...] en mémoire centrale.

Le processus de partionnement décrit ci-haut (appelé aussi segmentation) est le point central du tri rapide, nous construisons une fonction **Partition** réalisant cette action .

Comme l'on appliquant la même action sur les deux sous-listes obtenues après partition, la méthode est donc récursive, le tri rapide est alors une procédure récursive.

B-1) Voici une spécification générale de la procédure de tri rapide :

Tri Rapide sur [a..b]
Partition [a..b] renvoie **pivot** & [a..b] = **[x .. pivot']**+**[pivot]**+**[pivot'' .. y]**
Tri Rapide sur **[pivot'' .. y]**
Tri Rapide sur **[x .. pivot']**

B-2) Voici une spécification générale de la fonction de partionnement :

La fonction de partionnement d'une liste [a..b] doit répondre aux deux conditions suivantes :

- renvoyer la valeur de l'indice noté **i** d'un élément appelé pivot qui est bien placé définitivement : pivot = T[i],
- établir un réarrangement de la liste [a..b] autour du pivot tel que :

[a..b] = **[x .. pivot']**+**[pivot]**+**[pivot'' .. y]**

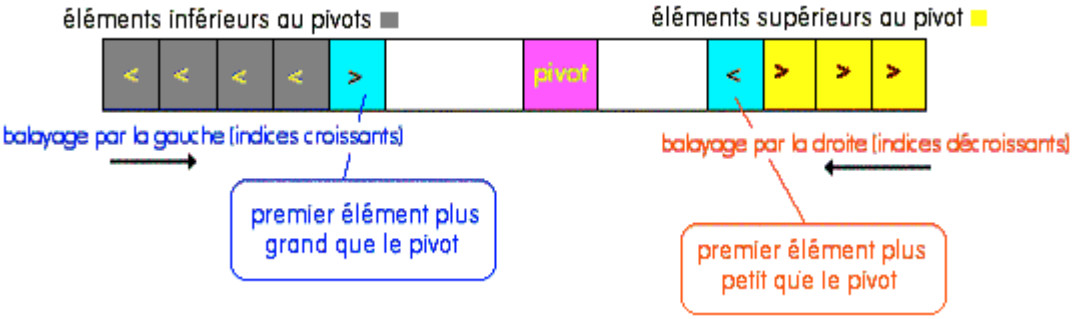
[x .. pivot'] = T[G] , .. , T[i-1]
(où : x = T[G] et pivot' = T[i-1]) tels que les T[G] , .. , T[i-1] sont tous inférieurs à T[i] ,

[pivot'' .. y] = T[i+1] , .. , T[D]
(où : y = T[D] et pivot'' = T[i+1]) tels que les T[i+1] , .. , T[D] sont tous supérieurs à T[i] ,

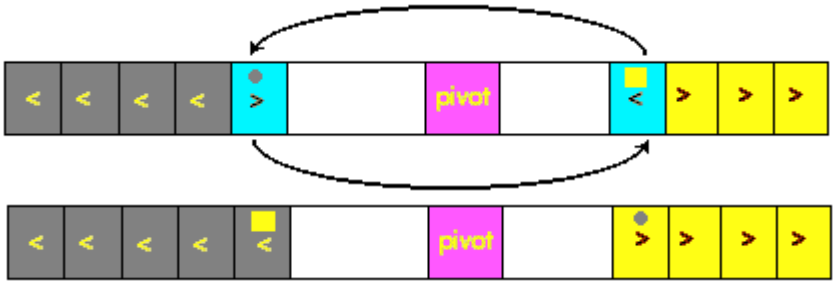
Il est proposé de **choisir arbitrairement le pivot** que l'on cherche à placer, puis ensuite de balayer la liste à réarranger dans les deux sens (par la gauche et par la droite) en construisant une sous-liste à gauche dont les éléments ont une valeur inférieure à celle du pivot et une sous-liste à droite dont les éléments ont une valeur supérieure à celle du pivot .

1) Dans le balayage par la gauche, on ne touche pas à un élément si sa valeur est inférieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous-liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus grande que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous-liste mais plutôt dans l'autre sous-liste.

2) Dans le balayage par la droite, on ne touche pas à un élément si sa valeur est supérieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous-liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus petite que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous-liste mais plutôt dans l'autre sous-liste.



3) on procède à l'échange des deux éléments mal placés dans chacune des sous-listes :



4) On continue le balayage par la gauche et le balayage par la droite tant que les éléments sont bien placés (valeur inférieure par la gauche et valeur supérieure par la droite), en échangeant à chaque fois les éléments mal placés.

5) La construction des deux sous-listes est terminée dès que l'on atteint (ou dépasse) le pivot.

Appliquons cette démarche à l'exemple précédent : L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]

- Choix arbitraire du pivot : l'élément le plus à droite ici **16**

- Balayage à gauche :
  4 < 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 16**]
  23 > 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage gauche,
  liste en cours de construction :[**4, 23, 16**]
- Balayage à droite :
  38 > 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 23, 16, 38**]
  18 > 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 23, 16, 18, 38**]
  45 > 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 23, 16, 45, 18, 38**]
  14 < 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage droit,
  liste en cours de construction : [**4, 23, 16, 14, 45, 18, 38**]

- Echange des deux éléments mal placés :

  [4, 23 , 16, 14 , 45, 18, 38] ----> [4, 14 , 16, 23, 45, 18, 38]

- On reprend le balayage gauche à l'endroit où l'on s'était arrêté :
  ----------↓----------
  [4, 14, 3, 42, 2, 23, 45, 18, 38, 16]
  3 < 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [4, 14, 3 , 16 , 23, 45, 18, 38]
  42 > 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête de nouveau le balayage gauche,
  liste en cours de construction : [4, 14, 3, 42 , 16, 23, 45, 18, 38]

- On reprend le balayage droit à l'endroit où l'on s'était arrêté :
  ----------↓----------
  [4, 14, 3, 42, 2, 23, 45, 18, 38, 16]

  2 < 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage droit,
  liste en cours de construction : [4, 14, 3, 42 , 16, 2, 23, 45, 18, 38]

- On procède à l'échange des deux éléments mal placés :
  [4, 14, 3, 42, 16, 2 , 23, 45, 18, 38] ----> [4, 14, 3, 2 , 16, 42, 23, 45, 18, 38]
  et l'on arrête la construction puisque nous sommes arrivés au pivot la fonction partition a terminé son travail elle a évalué :

  - le pivot : 16
  - la sous-liste de gauche : **L1 = [4, 14, 3, 2]**
  - la sous-liste de droite : **L2 = [23, 45, 18, 38, 42]**
  - la liste réarrangée : [4, 14, 3, 2, 16, 42, 23, 45, 18, 38]

  Il reste à recommencer les mêmes opérations sur les parties L1 et L2 jusqu'à ce que les partitions ne contiennent plus qu'un seul élément.

C) Algorithme :

```
Global :Tab[min..max] tableau d'entier

fonction Partition( G , D : entier )  résultat : entier
Local : i , j ,  piv , temp : entier
début
    piv  ← Tab[D];
    i  ← G-1;
    j  ← D;
    repeter
      repeter  i ← i+1 jusquà Tab[i] >= piv;
```

```
      repeter  j ← j-1 jusquà Tab[j] <= piv;
      temp ←  Tab[i];
      Tab[i] ← Tab[j];
      Tab[j] ←  temp
    jusquà j <= i;
    Tab[j] ←  Tab[i];
    Tab[i] ←  Tab[d];
    Tab[d] ←  temp;
    résultat ←  i
FinPartition

Algorithme TriRapide( G  , D : entier );
Local :  i : entier
début
 si D > G alors
   i ←Partition( G , D );
  TriRapide( G , i-1 );
  TriRapide( i+1 , D );
 Fsi
FinTRiRapide
```

Nous supposons avoir mis une sentinelle dans le tableau, dans la première cellule la plus à gauche, avec une valeur plus petite que n'importe qu'elle autre valeur du tableau.

Cette sentinelle est utile lorsque le pivot choisi aléatoirement se trouve être le plus petit élément de la table /pivot = min (**a**1, **a**2, ... , **a**n)/ :

Comme nous avons:
∀j , Tab[j] > piv , alors la boucle :

"**repeter** j ← j-1 **jusquà** Tab[j] <= piv ;"
pourrait ne pas s'arrêter ou bien s'arrêter sur un message d'erreur.

La sentinelle étant plus petite que tous les éléments y compris le pivot arrêtera la boucle et encore une fois évite de programmer le cas particulier du pivot = min (**a**1, **a**2, ... , **a**n).

D) Complexité :

Nous donnons les résultats classiques et connus mathématiquement (pour les démonstrations nous renvoyons aux ouvrages de R.Sedgewick & Aho-Ullman cités dans la bibliographie).

Choix opération

L'opération élémentaire choisie est **la comparaison de deux cellules** du tableau.

Comme tous les algorithmes qui divisent et traitent le problème en deux sous-problèmes le nombre moyen de comparaisons est en **O(n.log(n))** que l'on nomme **complexité moyenne.** La notation **log** (x) est utilisée pour le logarithme à base 2, $\log_2$ (x).

L'expérience pratique montre que cette complexité moyenne en **O(n.log(n))** n'est atteinte que lorsque les pivots successifs divisent la liste en deux sous-listes de taille à peu près équivalente.
Dans le pire des cas (par exemple le pivot choisi est systématiquement à chaque fois la plus grande valeur) on montre que la complexité est en $\mathbf{O(n^2)}$.

Comme la littérature a montré que ce tri était le meilleur connu en complexité, il a été proposé beaucoup d'améliorations permettant de choisir un pivot le meilleur possible, des combinaisons avec d'autres tris par insertion généralement, si le tableau à trier est trop petit….

Ce tri est pour nous un excellent exemple en **n.log(n)**.illustrant la récursivité.

E) Programme Delphi (tableau d'entiers) :

```
program TriQuickSort;

const N = 10;  { Limite supérieure de tableau }
type TTab = array [0..N] of integer;   { TTab : Type Tableau }
var Tab : TTab ;

function Partition ( G , D : integer) : integer;
  var i , j : Integer;
      piv, temp  : integer;
begin
    i := G-1;
    j := D;
    piv := Tab[D];
    repeat
       repeat i := i+1 until Tab[i] >= piv;
       repeat j := j-1 until Tab[j] <= piv;
       temp :=Tab[i];
       Tab[i] :=Tab[j];
       Tab[j] :=temp;
    until j <= i;
    Tab[j] := Tab[i];
    Tab[i] := Tab[d];
    Tab[d] := temp;
    result := i;
end;{Partition}
```

```
procedure TriRapide( G, D : integer);
var i: Integer;
begin
  if D>G then
  begin
    i := Partition( G , D );
    TriRapide( G , i-1 );
    TriRapide( i+1 , D );
  end
end;{TriRapide}
```

```
procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
  Tab[0] := -Maxint ; // la sentinelle
end;

procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('-------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
       writeln;
end;

begin
  Initialisation(Tab);
  writeln('TRI RAPIDE');
  writeln;
  Impression(Tab);
  TriRapide( 1 , N );
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI RAPIDE

-------------------------------------------------------------
 17 |  32 |  14 |  45 |  54 |  50 |  60 |  10 |  68 |  12 |
-------------------------------------------------------------
 10 |  12 |  14 |  17 |  32 |  45 |  50 |  54 |  60 |  68 |
-------------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriQSort
{
 static int[] table = new int[21] ; // le tableau à trier en attribut
 /*  Les cellules [0] et [20] contiennent
     des sentinelles,
     Les cellules utiles vont de 1 à 19.
     (de 1 à table.length-2)
 */

 static void impression ( )
 {
  // Affichage sans les sentinelles
  int n = table.length-2;
  for ( int i = 1; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void initialisation  ( )
 {
  // remplissage aléatoire du tableau
  int n = table.length-2;
  for(int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
  // Les sentinelles:
  table[0] = -Integer.MAX_VALUE;
  table[n+1] = Integer.MAX_VALUE;
 }

 // ----> Tri rapide :
```

```
static int partition (int G, int D )
{ // partition / Sedgewick /
 int i, j, piv, temp;
 piv = table[D];
 i = G-1;
 j = D;
 do
  {
   do
    i++;
   while (table[i]<piv);
   do
    j--;
   while (table[j]>piv);
   temp = table[i];
   table[i] = table[j];
   table[j] = temp;
  }
 while(j>i);
 table[j] = table[i];
 table[i] = table[D];
 table[D] = temp;
 return i;
}
```

```
static void QSort (int G, int D )
{ // tri rapide, sous-programme récursif
 int i;
 if(D>G)
 {
  i = partition(G,D);
  QSort(G,i-1);
  QSort(i+1,D);
 }
}
```

```
 public static void main(string[ ] args)
 {
  Initialisation ( );
  int n = table.length-2;
  System.out.println("Tableau initial :");
  Impression ( );
  QSort(1,n);
  System.out.println("Tableau une fois trié :");
  Impression ( );
 }
}
```

Le tri par arbre

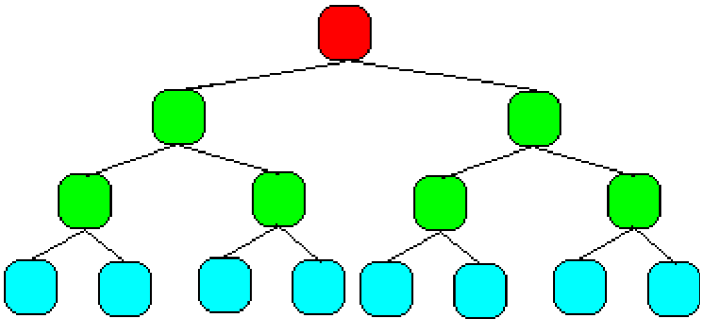
- Définitions préliminaires
- **A)** Spécification abstraite
- **B)** Spécification concrète
- **C)** Algorithme
- **D)** Complexité
- **E)** Programme Delphi

C'est un tri également appelé tri par tas (*heapsort*, en anglais). Il utilise une structure de données temporaire dénommée "tas" comme mémoire de travail.

Définitions préliminaires

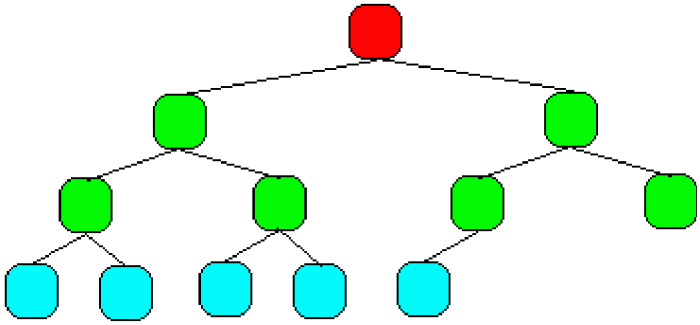
Définition - 1 / Arbre parfait :

c'est un arbre binaire dont tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles), dans ce cas l'arbre parfait est un arbre binaire incomplet et les feuilles du dernier niveau doivent être regroupées à partir de la gauche de l'arbre
.

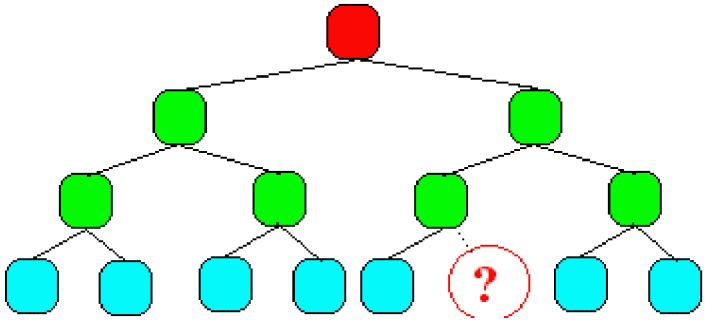


un arbre parfait complet

Amputons l'arbre parfait précédent de ses trois feuilles situées sur le bord droit, les cinq premières feuilles de gauche ne changeant pas, on obtient toujours un arbre parfait mais il est incomplet :



un autre arbre parfait incomplet

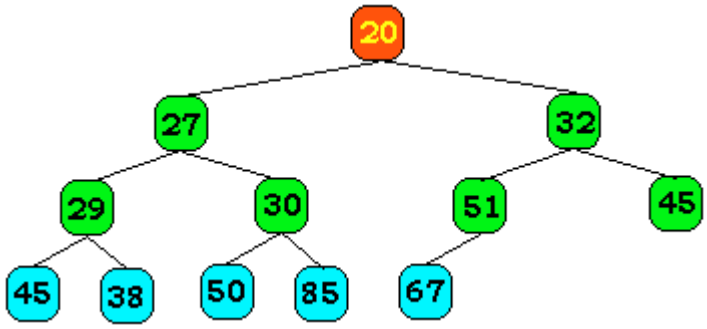


exemple d'arbre non parfait

Définition - 2 / Arbre partiellement ordonné :

C'est un arbre étiqueté dont les noeuds appartiennent à un ensemble muni d'une relation d'ordre total (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses fils ont une valeur supérieure ou égale à celle de leur père.

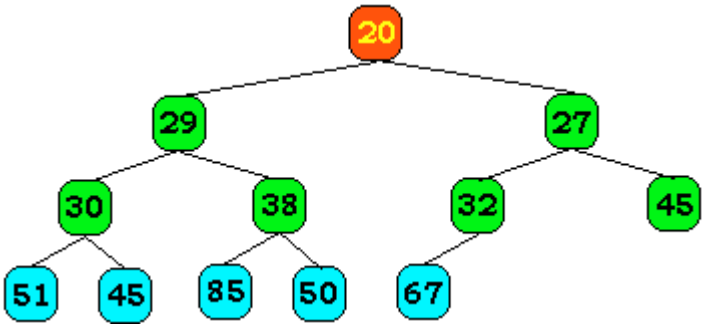
Exemple d'un arbre partiellement ordonné sur l'ensemble {20, 27,29, 30, 32, 38, 45, 45, 50, 51, 67 ,85 } d'entiers naturels :



Nous remarquons que **la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum** (le plus petit élément de l'ensemble), car la valeur de ce noeud par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses descendants c'est le minimum.

Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre.

Exemple d'un autre arbre partiellement ordonné sur le même ensemble {20, 27,29, 30, 32, 38, 45, 45, 50, 51, 67 ,85 } d'entiers naturels (il n'y a pas unicité) :



Définition - 3 / Le tas :

On appelle **tas** un tableau représentant un **arbre parfait partiellement ordonné**.

Principe du tri par tas

C'est une variante de méthode de tri par sélection où l'on parcourt le tableau des éléments en sélectionnant et conservant les minimas successifs (plus petits éléments partiels) dans un arbre parfait partiellement ordonné.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par tas est de parcourir la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en ajoutant chaque élément **a**k dans **un arbre parfait partiellement ordonné**.

- L'insertion d'un nouvel élément $\mathbf{a}_k$ dans l'arbre a lieu **dans la dernière feuille vide de l'arbre à partir de la gauche** (ou bien si le niveau est complet en recommençant un nouveau niveau par sa feuille la plus à gauche) et, en effectuant des échanges tant que la valeur de $\mathbf{a}_k$ est inférieur à celle de son pére.

- Lorsque tous les éléments de la liste seront placés dans l'arbre, l'élément minimum "$\mathbf{a}_i$" de la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) se retrouve à la racine de l'arbre qui est alors partiellement ordonné.

- On recommence le travail sur la nouvelle liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) -{$\mathbf{a}_i$}(c'est la liste précédente privée de son minimum),

  pour cela on supprime l'élément minimum **a**i de l'arbre pour le mettre dans la liste triée puis,

  on prend l'élément de la dernière feuille du dernier niveau et on le place à la racine.

  On effectue ensuite des échanges de contenu avec le fils dont le contenu est inférieur, en partant de la racine, et en descendant vers le fils avec lequel on a fait un échange, ceci tant qu'il n'a pas un contenu inférieur à ceux de ses deux fils (ou de son seul fils) ou tant qu'il n'est pas à une feuille.

- On recommence l'opération de suppression et d'échanges éventuels **jusqu'à ce que l'arbre ne contienne plus aucun élément**.

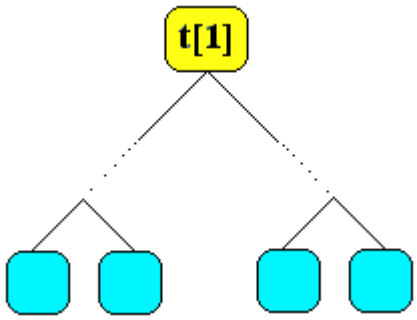
B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimension T[...] correspondant au tableau d'initialisation. Puis les éléments de ce tableau sont ajoutés et traités un par un dans un arbre avant d'être ajoutés dans un tableau trié en ordre décroissant ou croissant, selon le choix de l'utilisateur.

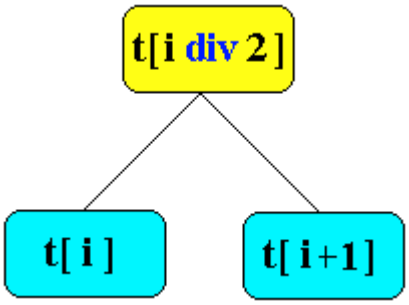
Signalons qu'un arbre binaire parfait se représente classiquement par un tableau :

Si t est ce tableau, on a les règles suivantes :

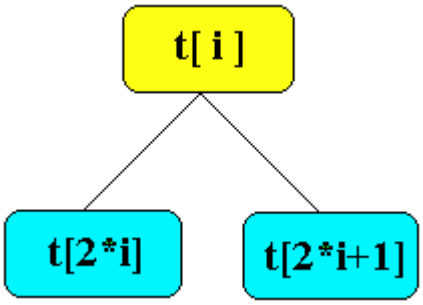
- t[1] est la racine :



- t[i **div** 2] est le père de t[i] pour i > 1 :



- t[2 * i] et t[2 * i + 1] sont les deux fils, s'ils existent, de t[i] :



- si **p** est le nombre de noeuds de l'arbre et si 2 * i = p, t[i] n'a qu'un fils, t[p].
  si i est supérieur à p **div** 2, t[i] est une feuille.

Figures illustrant le placement des éléments de la liste dans l'arbre

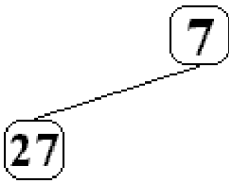
Exemple d'initialisation sur un tableau à 15 éléments :

7	27	41	30	10	31	22	33	23	17	3	25	44	7	25

Insertion du premier élément (le nombre **7**) à la racine de l'arbre :

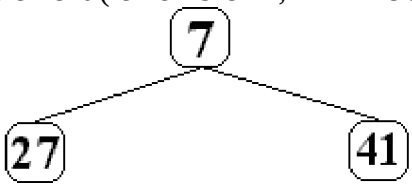
	27	41	30	10	31	22	33	23	17	3	25	44	7	25

Insertion du second élément (le nombre **27**, 27 > 7 donc c'est un fils du noeud 7) :



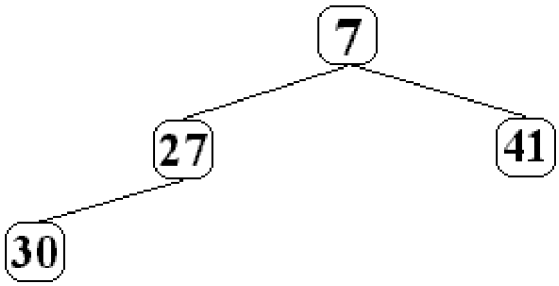
		41	30	10	31	22	33	23	17	3	25	44	7	25

Insertion du troisième élément (le nombre **41**, 41 > 7 c'est un fils du noeud 7) :



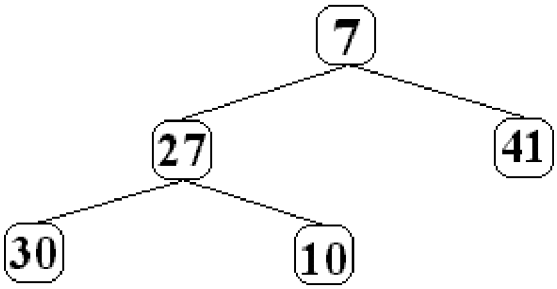
			30	10	31	22	33	23	17	3	25	44	7	25

Insertion du quatrième élément (le nombre **30**, comme le niveau des fils du noeud 7 est complet, 30 est placé automatiquement sur une nouvelle feuille d'un nouveau niveau, puis il est comparé à son père 27, 30 > 27 c'est donc un fils du noeud 27 il n'y a pas d'échange) :

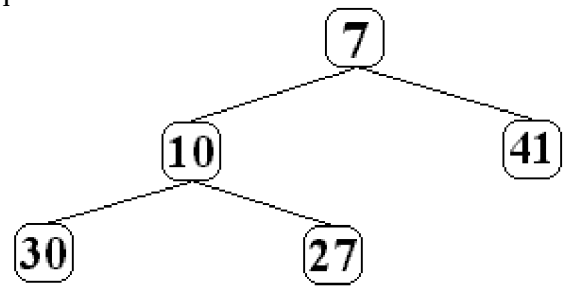


				10	31	22	33	23	17	3	25	44	7	25

Insertion du cinquième élément (le nombre 10) : L'insertion du nouvel élément 10 dans l'arbre a lieu automatiquement dans la dernière feuille vide de l'arbre à partir de la gauche, ici le fils droit de 27 :



Puis 10 est comparé à son père 27, cette fois 10 est plus petit que 27, il y a donc échange des places entre 27 et 10, ce qui donne un nouvel arbre :

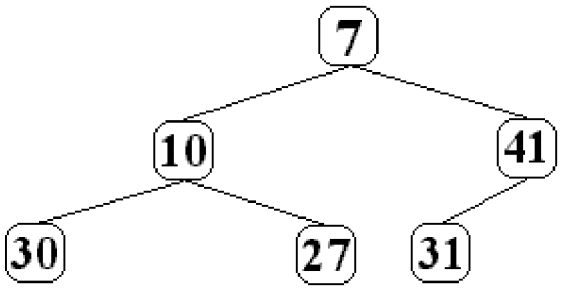


Puis 10 est comparé à son nouveau père 7, cette fois il n'y a pas d'échange car 10 est plus grand que 7.

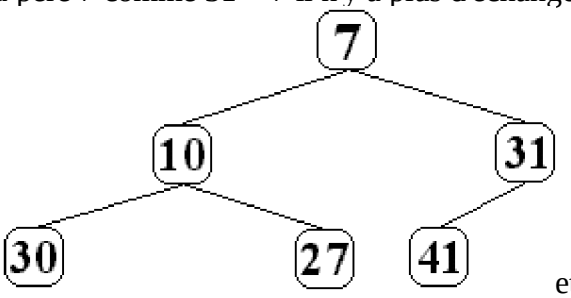
Le processus continue avec l'élément suivant de la liste le nombre 31:

					31	22	33	23	17	3	25	44	7	25

31 est automatiquement rangé sur la première feuille disponible à gauche soit le fils gauche de 41:



Puis 31 est comparé à son père, comme 31 < 41 il y a échange des valeurs, puis 31 est comparé à son nouveau père 7 comme 31 > 7 il n'y a plus d'échange :

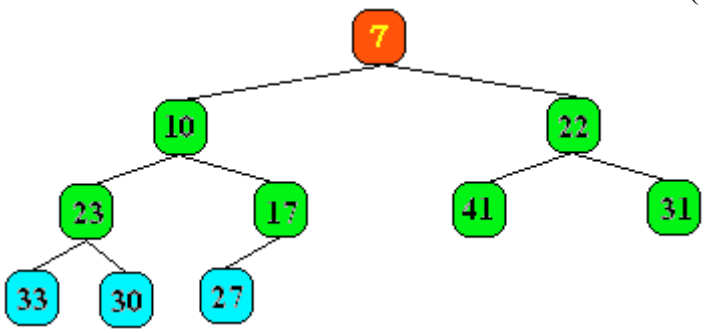


etc....

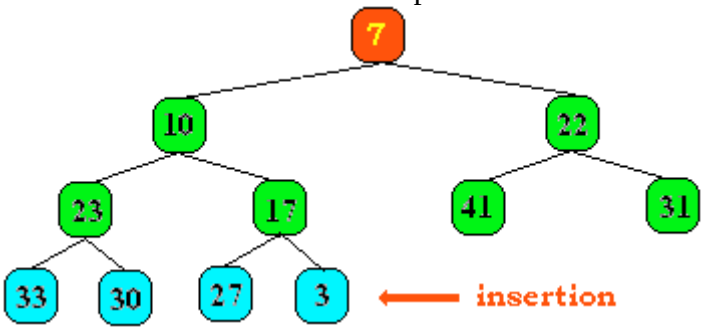
Supposons que l'arbre ait été construit sur les dix premiers éléments du tableau et observons maintenant comment l'élément minimum de la liste qui est le onzième élément, soit le nombre 3, est rangé dans l'arbre.

										3	25	44	7	25

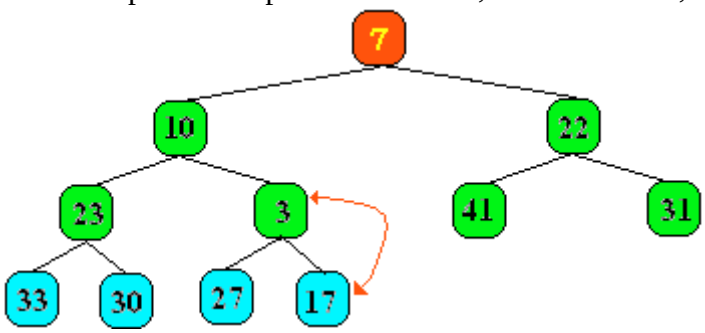
Voici l'état de l'arbre avant introduction du nombre 3 (quatre niveaux de nœuds) :



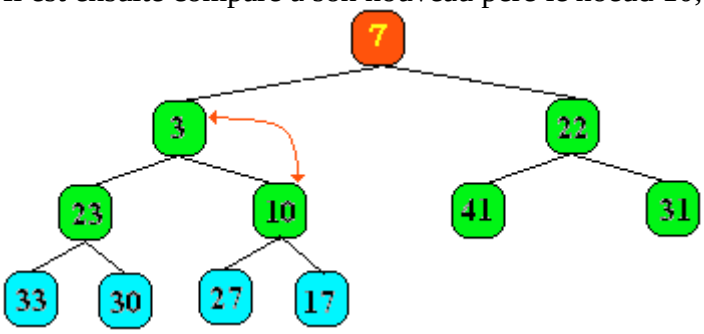
Le nombre 3 est introduit sur la première feuille libre du niveau quatre :



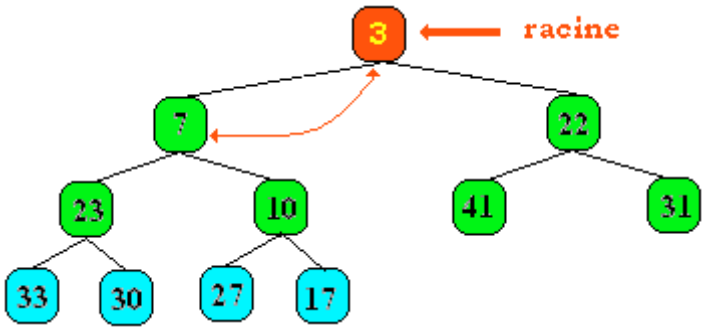
Il est comparé à son père le noeud 17, comme 3 < 17, il y a alors échange :



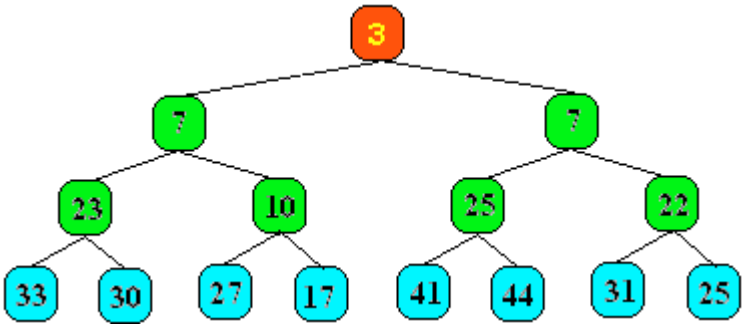
Il est ensuite comparé à son nouveau père le noeud 10, comme 3 < 10, il y a alors échange :



Il est enfin comparé à son dernier père (la racine de l'arbre), comme 3 < 7, il y a alors échange :



C'est l'élément **3** qui est maintenant **la racine** de l'arbre, comme les 4 éléments suivants sont plus grand que 3, ils seront rangés plus bas dans l'arbre (cf. figure ci-dessous) :

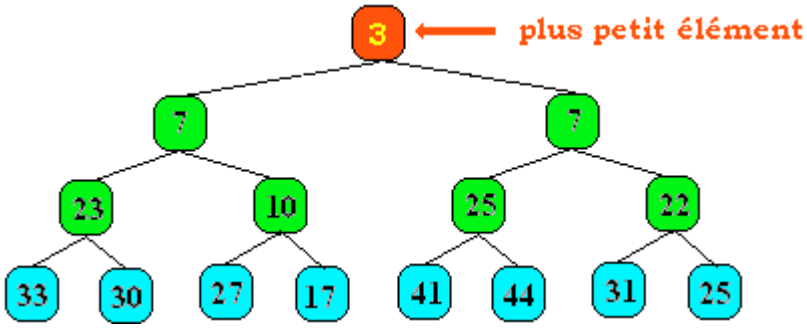


Conclusion sur le premier passage :

La liste initiale :

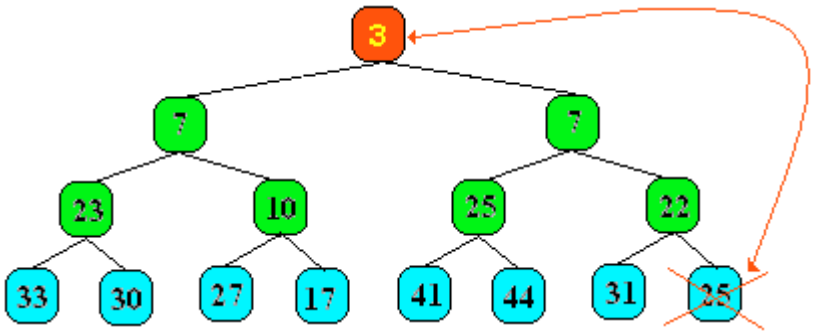
7	27	41	30	10	31	22	33	23	17	3	25	44	7	25

est finalement stockée ainsi :

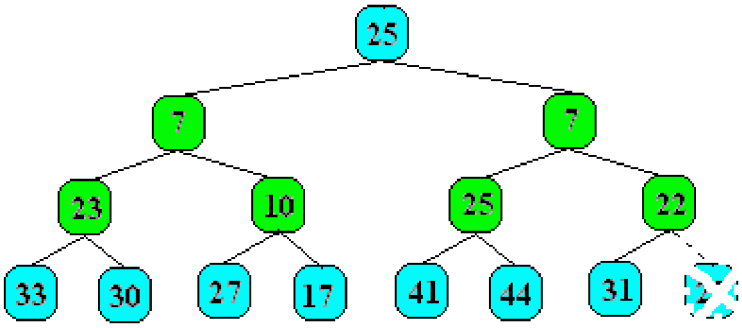


Figures illustrant la suppression de la racine

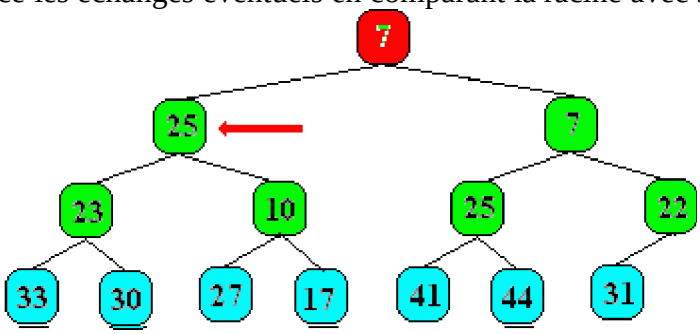
Le nombre **3** est le plus petit élément, on le supprime de l'arbre et l'on prend l'élément de la dernière feuille du dernier niveau (ici le nombre 25) et on le place à la racine (à la place qu'occupait le nombre 3)



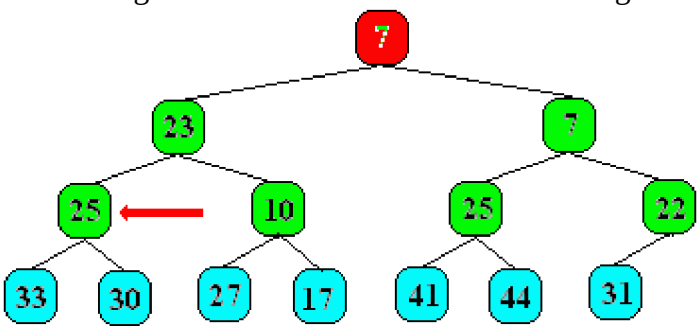
ce qui donne comme nouvelle disposition :



et l'on recommence les échanges éventuels en comparant la racine avec ses descendants :



le fils gauche 23 est inférieur à 25 => échange



Arrêt du processus (33 > 25 et 30 > 25)

On obtient le deuxième plus petit élément à la racine de l'arbre, ici le nombre 7. Puis l' on continue à "vider" ainsi l'arbre et déplaçant les feuilles vers la racine et en échangeant les noeuds mal placés.

A la fin, lorsque l'arbre a été entièrement vidé, nous avons extrait à chaque étape le plus petit élément de chaque sous liste restante et nous obtenons les éléments de la liste classés par ordre croissant ou décroissant selon notre choix (dans notre exemple si nous stockons les minima successifs de gauche à droite dans une liste nous obtiendrons une liste classée par ordre croissant de gauche à droite).

En résumé notre version de tri par tas comporte les étapes suivantes :

- initialisation : ajouter successivement chacun des n éléments dans le tas t[1..p]; p augmente de 1 après chaque adjonction . A la fin on a un tas de taille p = n.

- tant que p est plus grand que 1, supprimer le minimum du tas (p décroît de 1), réorganiser le tas, ranger le minimum obtenue à la $(p + 1)^{\text{ième}}$ place.

On en déduit l'algorithme ci-dessous composé de 2 sous algorithmes Ajouter pour la première étape, et Supprimer pour la seconde.

C) Algorithme :

```
Algorithme Ajouter
Entrée  P : entier ; x : entier // P nombre d'éléments dans le tas, x élément à ajouter
          Tas[1..max]  : tableau d'entiers // le tas
Sortie  P : entier
          Tas[1..max] // le tas
Local  j, temp : entiers
début
  P ← P + 1 ; // incrémentation du nombre d'éléments du tas
  j ← P ; // initialisation de j à la longueur du tas (position de la dernière feuille)
  Tas[P] ← x ; // ajout l'élément x à la dernière feuille dans le tas
  Tantque (j > 1) et (Tas[j] < Tas[j div 2]) faire ; // tant que l'on est pas arrivé à la racine et
que le "fils" est inférieur à son "père", on permute les 2 valeurs
        temp ← Tas[j] ;
        Tas[j] ← Tas[j div 2] ;
        Tas[j div 2] ← temp ;
        j ← j div 2 ;
  finTant
FinAjouter

Algorithme Supprimer
Entrée : P : entier // P nombre d'éléments contenu dans l'arbre
          Tas[1..max]  : tableau d'entiers // le tas
Sortie : P : entier // P nombre d'éléments contenu dans l'arbre
          Tas[1..max]  : tableau d'entiers // le tas
          Lemini : entier  // le plus petit de tous les éléments du tas
Local i, j, temp : entiers ;
début
  Lemini ← Tas[1] ; // retourne  la racine (minimum actuel) pour stockage éventuel
  Tas[1] ←Tas[P] ; // la racine prend la valeur de la dernière feuille de l'arbre
  P ← P - 1 ;
  j ←1 ;
  Tantque j <= (P div 2) faire
     // recherche de l'indice i du plus petit des descendants de Tas[j]
     si (2 * j = P ) ou (Tas[2 * j] < Tas[2 * j + 1])
          alors i ← 2 * j ;
     sinon i ←2 * j +1 ;
     Fsi;
    // Echange éventuel entre Tas[j] et son fils Tas[i]
    si Tlab[j] > Tlab[i] alors
          temp ← Tlab[j] ;
          Tlab[j] ← Tlab[i] ;
          Tlab[i] ← temp ;
          j ← i ;
    sinon Sortir ;
    Fsi;
  finTant
FinSupprimer
```

```
Algorithme Tri_par_arbre
Entrée  Tas[1..max] // le tas
        TabInit[1..max] // les données initiales
Sortie  TabTrie[1..max]: tableaux d'entiers //  tableau une fois trié
Local   P  : Entier //  P le nombre d'éléments à trier
        Lemin : entier  // le plus petit de tous les éléments du tas
début
   P ← 0;
  Tantque P < max faire
     Ajouter(P,Tas,TabInit[P+1]); // appel de l'algorithme Ajouter
  finTant;
  Tantque P >= 1 faire
     Supprimer(P,Tas,Lemin) ; // appel de l'algorithme Supprimer
     TabTrie[max-P] ← Lemin ; // stockage du minimum dans le nouveau tableau
  finTant;
Fin Tri_par_arbre
```

D) Complexité :

Cette version de l'algorithme construit le tas par n appels de la procédure **Ajouter** et effectue les sélections par n - 1 appels de la procédure supprimer.

Le coût et de l'ordre de $\sum_{i=1}^{n} \log_2 i$ comparaisons, au pire.

La complexité au pire en nombre de comparaisons est en O(n log n).

Le nombre d'échanges dans le tas est majoré par le nombre de comparaisons et il est du même ordre de grandeur.

La complexité au pire en nombre de transferts du tri par tas est donc en O(n log n).

E) Programme Delphi (tableau d'entiers) :

```
program TriParArbre;
const Max =10; // nombre maximal d'éléments du tableau
type TTab=array [1..Max] of integer;  // TTab : Type Tableau
var Tas, TabInit, TabTrie : TTab; // Tas, tableau initial puis tableau
triéTableau
     P, Lemin : integer;

procedure Ajouter (var Tas : TTab; var P, x : integer);
  var j, temp : integer ;
```

```
begin
   P := P + 1 ;
   J := P ;
   Tas[P] := x ;
   if j>1 then
   While Tas[j] < Tas[j div 2] do
    begin
       temp := Tas[j] ;
       Tas[j] := Tas[j div 2] ;
       Tas[j div 2] := temp ;
       j := j div 2 ;
      if j<=1 then break;
    end
end; // Ajouter

procedure Supprimer (var Tas:TTab; var P, Lemin : integer);
 var i, j, temp : integer ;
 begin
   Lemin := Tas[1] ;
   Tas[1] := Tas[P] ;
   P := P - 1 ;
   j := 1 ;
   While j <= (P div 2) do
   begin
     if (2 * j = P ) or (Tas[2 * j] < Tas[2 * j + 1])
         then i := 2 * j
     else i := 2 * j +1 ;
     if Tas[j] > Tas[i] then
     begin
       temp := Tas[j] ;
       Tas[j] := Tas[i] ;
       Tas[i] := temp ;
       j := i ;
     end
     else break ;
   end
end; // Supprimer

procedure Initialisation(var Tab:TTab) ;
// Tirage aléatoire de Max nombres de 1 à 100
var i : integer;  // i : Indice de tableau
begin
  randomize;
  for i := 1 to Max do
     Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
// Affichage des Max nombres
var i : integer;
```

```
begin
  writeln('----------------------');
  for i:= 1 to Max do write(Tab[i] : 3, ' | ');
      writeln;
end;

begin // TriParArbre
  Initialisation(TabInit);
  writeln('TRI PAR ARBRE');
  writeln;
  Impression(TabInit);
  P:=0;
  while p < Max do
    Ajouter ( Tas, p, TabInit[p+1] );
  while p >= 1 do
  begin
     Supprimer ( Tas, P, Lemin ) ;
     TabTrie[Max-P]:=Lemin
  end;
  Impression(TabTrie);
  writeln('----------------------');
  readln
end. // TriParArbre
```

REMARQUE IMPORTANTE

Notons que dans la procédure nous avons traduit la condition de la boucle :

```
Tantque (j > 1) et (Tas[j] < Tas[j div 2]) faire
        temp ← Tas[j] ;
        Tas[j] ← Tas[j div 2] ;
        Tas[j div 2] ← temp ;
        j ← j div 2 ;
 finTant
```

par les lignes de programmes suivantes :

```
if j>1 then
   While Tas[j] < Tas[j div 2] do
    begin
        temp := Tas[j] ;
        Tas[j] := Tas[j div 2] ;
        Tas[j div 2] := temp ;
        j := j div 2 ;
       if j<=1 then break
    end
```

ceci afin d'éviter un incident dû à un effet de bord. Lorsque l'indice "j" prend par exemple la valeur 1, l'indice "j **div** 2" vaut alors **0** et cette valeur n'est pas valide puisque l'indice de tableau varie entre **1**..**Max**.

On peut pallier aussi d'une autre manière à cet inconvénient en ajoutant une **sentinelle** "à gauche dans le tableau" en étendant la borne inférieure à la valeur **0**, les indices pouvant alors varier entre **0**..**Max**. On obtient une autre écriture de la procédure "Ajouter" qui suit malgré tout l'algorithme de près :

```
type TTab=array [0..Max] of integer;   // 0 est l'indice sentinelle

    procedure Ajouter (var Tas : TTab; var P, x : integer);
      var j, temp : integer ;
    begin
       P := P + 1 ;
       J := P ;
       Tas[P] := x ;
       While (j > 1) et (Tas[j] < Tas[j div 2]) do
        begin
           temp := Tas[j] ;
           Tas[j] := Tas[j div 2] ;
           Tas[j div 2] := temp ;
           j := j div 2 ;
        end
    end; // Ajouter
```

Résultat de l'exécution du programme précédent :

```
TRI PAR ARBRE

-------------------------------------------------------------
 93 |  74 |  13 |   1 |  14 |  22 |  42 |  99 |  16 |  48 |
-------------------------------------------------------------
  1 |  13 |  14 |  16 |  22 |  42 |  48 |  74 |  93 |  99 |
-------------------------------------------------------------
```

3. Rechercher dans un tableau

Les tableaux sont des structures statiques contiguës, il est facile de rechercher un élément fixé dans cette structure. Nous exposons ci-après des algorithmes élémentaires de recherche utilisables dans des applications pratiques par le lecteur.

Essentiellement nous envisagerons la **recherche séquentielle** qui convient lorsqu'il y a peu d'éléments à consulter (quelques centaines), et la **recherche dichotomique** dans le cas où la liste est triée.

3.1 Recherche dans un tableau non trié

Algorithme de recherche séquentielle :

- Soit **t** un tableau d'entiers de **1..n** éléments non rangés.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)
- Complexité en O(n)

Nous proposons au lecteur 4 versions d'un même algorithme de recherche séquentielle. Le lecteur adoptera une de ces versions en fonction des possibilités du langage avec lequel il compte programmer sa recherche.

Version **Tantque** avec "**et alors**" (si le langage dispose d'un opérateur **et** optimisé)	Version **Tantque** avec "**et**" (opérateur **et** non optimisé)

i ← 1 ;
Tantque (i ≤ n) **et alors** (t[i] ≠ Elt) **faire**
  i ← i+1
finTant;
si i ≤ n **alors** rang ← i
sinon rang ← -1
Fsi
	i ← 1 ;
	Tantque (i < n) **et** (t[i] ≠ Elt) **faire**
	  i ← i+1
	finTant;
	si t[i] = Elt **alors** rang ← i
	sinon rang ← -1
	Fsi

Version **Tantque** avec sentinelle en fin de tableau (rajout d'une cellule supplémentaire en fin de tableau contenant systématiquement l'élément à rechercher)

Version **Tantque** avec sentinelle avec "**et alors**"	Version **Pour** avec instruction de sortie (conseillée si le langage dispose d'un opérateur **Sortirsi**)
t[n+1] ← Elt ; *// sentinelle rajoutée* i ← 1 ; **Tantque** (i ≤ n) **et alors** (t[i] ≠ Elt) **faire** i ← i+1 **finTant;** **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**	**pour** i ← 1 **jusquà** n **faire** **Sortirsi** t[i] = Elt **fpour**; **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**

3.2 Recherche dans un tableau trié

Spécifications d'un algorithme séquentiel

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**) ***On peut reprendre sans changement les versions de l'algorithme de recherche séquentielle précédent travaillant sur un tableau non trié.***
- Complexité moyenne en O(n)

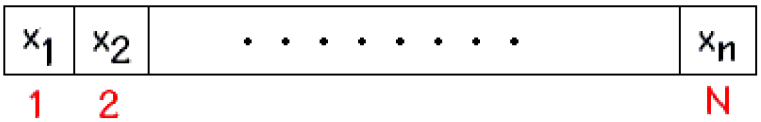
On peut aussi utiliser le fait que le dernier élément du tableau est **le plus grand élément** et s'en servir comme une sorte de **sentinelle**. Ci-dessous deux versions utilisant cette dernière remarque.

Version **Tantque**	Version **pour**
si t[n] < Elt **alors** rang ← -1 **sinon** i ← 1 ; **Tantque** t[i] < Elt **faire** i ← i+1; **finTant;** **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi** **Fsi**	**si** t[n] < Elt **alors** rang ← -1 **sinon** **pour** i ← 1 **jusquà** n-1 **faire** **Sortirsi** t[i] ≥ Elt *// sortie de la boucle* **fpour**; **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi** **Fsi**

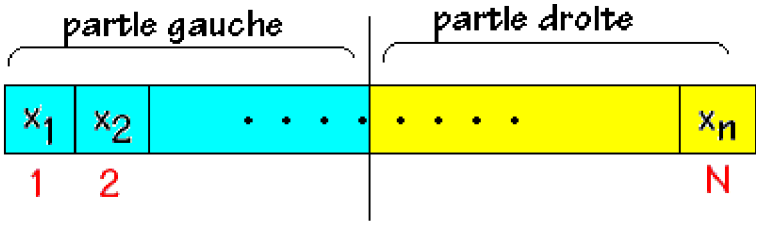
Spécifications d'un algorithme dichotomique

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**). **Au lieu de rechercher séquentiellement du premier jusqu'au dernier, on compare l'élément Elt à chercher au contenu du milieu du tableau. Si c'est le même, on retourne le rang du milieu, sinon l'on recommence sur la première moitié (ou la deuxième) si l'élément recherché est plus petit (ou plus grand) que le contenu du milieu du tableau.**
- Complexité moyenne en O(log(n))

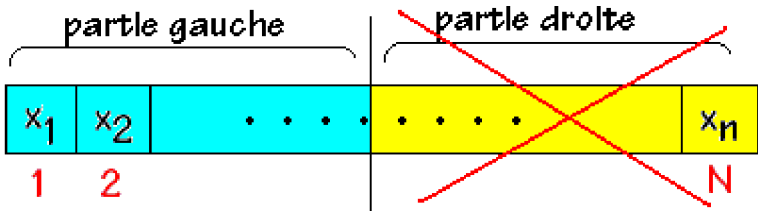
Soit un tableau au départ contenant les éléments (x_1, x_2, ... , x_n)



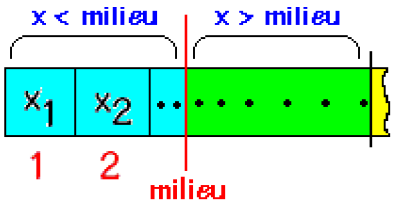
On recherche la présence de l'élément x dans ce tableau. On divise le tableau en 2 parties égales :



Soit x est inférieur à l' élément milieu et s'il est présent il ne peut être que dans la partie gauche, soit x est supérieur à l' élément milieu et s'il est présent il ne peut être que dans la partie droite. Supposons que x < milieu nous abandonnons la partie droite et nous recommençons la même division sur la partie gauche :



On divise à nouveau la partie gauche en deux parties égales avec un nouveau milieu :



Si x < milieu c'est la partie gauche qui est conservée sinon c'est la partie droite etc ...
Le principe de la division dichotomique aboutit à la fin à une dernière cellule dans laquelle soit x = milieu et x est présent dans la table, soit x ≠ milieu et x n'est pas présent dans la table.

Version itérative du corps de cet algorithme :

```
bas, milieu, haut, rang : entiers

    bas ← 1;
    haut ← N;
    Rang ← -1;
    repéter
      milieu ← (bas + haut) div 2;
      si x = t[milieu] alors
         Rang ← milieu
      sinon si t[milieu] < x alors
                bas ← milieu + 1
            sinon
                haut ← milieu-1
            fsi
      fsi
    jusquà ( x = t[milieu] ) ou ( bas haut )
```

Voici en Delphi une procédure traduisant la version itérative de cet algorithme :

```
procedure dichoIter(x:Elmt; table:tableau; var rang:integer);
{recherche dichotomique par itération dans table rang =-1 si pas trouvé}
var
  milieu:1..max;
  g,d:0..max+1;
begin
 g := 1;
 d := max;
 rang := -1;
 while g <= d do
 begin
  milieu := (g+d) div 2;
  if x = table[milieu] then
  begin
    rang := milieu;
    exit
  end
  else
    if x < table[milieu] then d := milieu-1
    else g := milieu+1
 end;
end;{dichoIter}
```

Dans le cas de langage de programmation acceptant la récursivité (comme Delphi), il est possible d'écrire une version récursive de cet algorithme dichotomique.

Voici en Delphi une procédure traduisant la version récursive de cet algorithme :

```
procedure dichoRecur(x:Elmt;table: tableau; g,d:integer; var rang:integer);
{ recherche dichotomique récursive dans table
   rang =-1 si pas trouvé.
   g , d: 0..max+1 }
var
   milieu:1..max;
begin
  if g<= d then
  begin
   milieu := (g+d) div 2;
   if x = table[milieu] then  rang := milieu
   else
     if x < table[milieu] then // la partie gauche est conservée
         dichoRecur( x, table, g, milieu-1, rang )
     else // la  partie droite est conservée
         dichoRecur( x, table, milieu+1, d, rang )
  end
  else rang := -1
end;{dichoRecur}
```

Exercices chapitre 3

Des exercices traités avec leur solution détaillée

Algorithmes et leur traduction en langage de programmation

- **Somme de 2 vecteurs**
- **Fonctions booléennes**
- **Variantes sur la factorielle**
- **PGCD , PPCM de deux entiers**
- **Nombres premiers**
- **Nombres parfaits**
- **Suite : racine carrée - Newton**
- **Inversion d'un tableau**

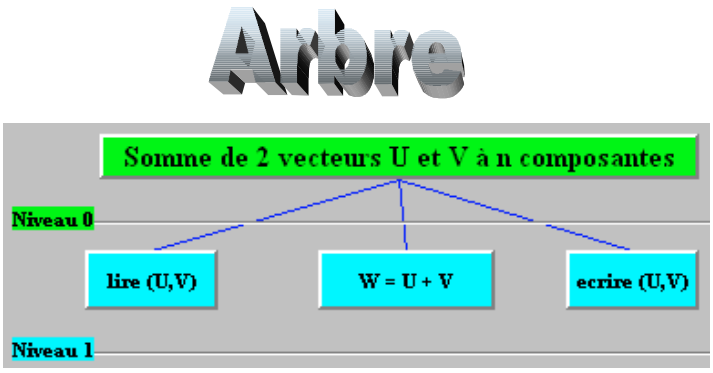
Des algorithmes

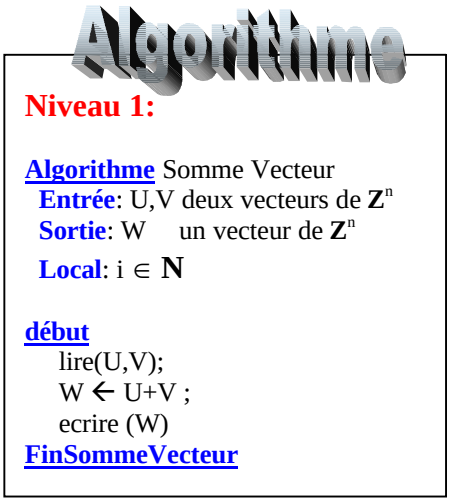
Somme de 2 vecteurs

Enoncé :

Programme simple d'utilisation des tableaux, on représente un vecteur de $\mathbf{Z}^n$ dans un tableau à un indice variant de 1 à n. Ecrire un programme LDFA additionnant 2 vecteurs de $\mathbf{Z}^n$ dont les composantes sont lues au clavier.

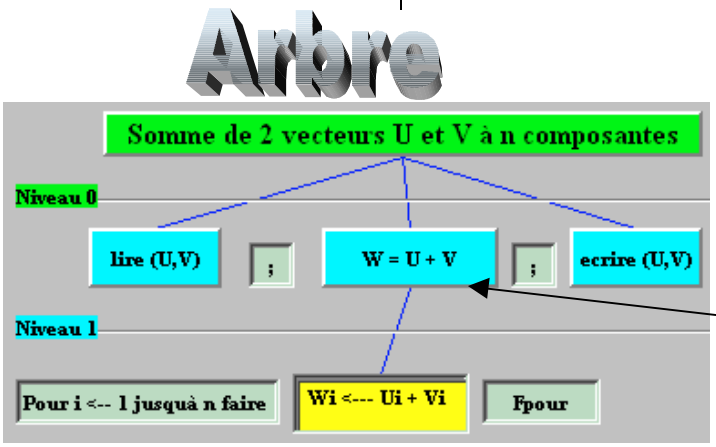
Solution faisant apparaître les niveaux de décomposition et l'algorithme associé

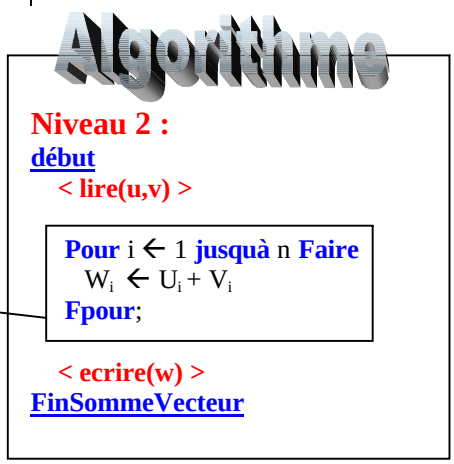


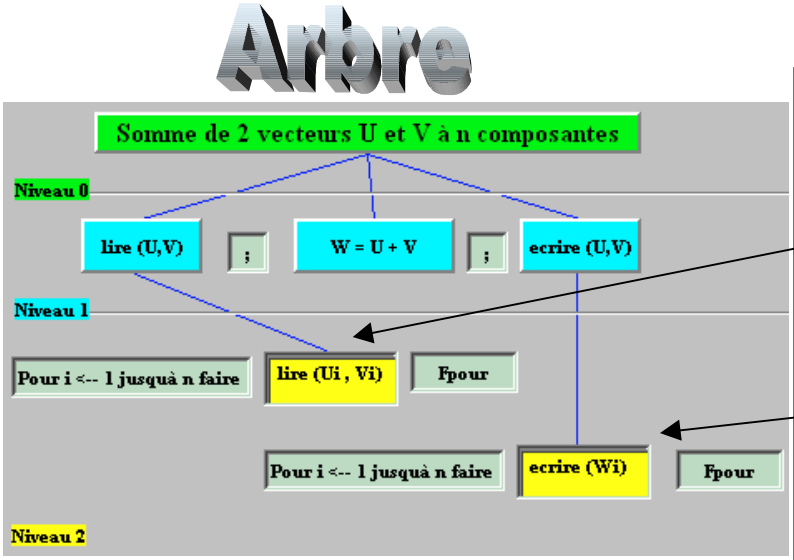


Un vecteur de Z^n est défini par ses coordonnées : $U(U_1, U_2, \ldots, U_n)$; $V(V_1, V_2, \ldots, V_n)$ etc…

Action < W = U+V > :	Action < Lire (U,V) > :	Action < Ecrire(W) > :
$\forall i, 1 \le i \le n$ / $W_i = U_i + V_i$	$\forall i, 1 \le i \le n$ / lire U_i et V_i	$\forall i, 1 \le i \le n$ / ecrire W_i
Description :	*Description* :	*Description* :
Pour i ← 1 **jusquà** n **Faire** $W_i \leftarrow U_i + V_i$ **Fpour**;	**Pour** i ← 1 **jusquà** n **Faire** lire(U_i , V_i) **Fpour**;	**Pour** i ← 1 **jusquà** n **Faire** ecrire(W_i) **Fpour**;







Algorithme

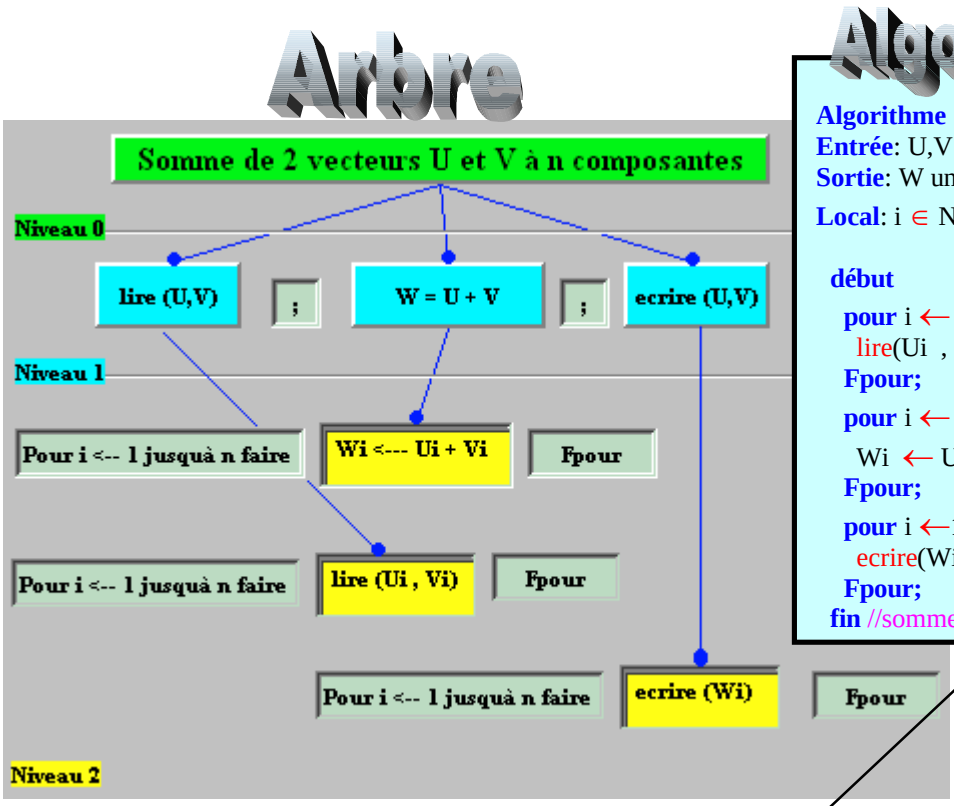
début

Pour i ← 1 **jusquà** n **Faire**
lire(U_i , V_i)
Fpour;

W = U+V

Pour i ← 1 **jusquà** n **Faire**
ecrire(W_i)
Fpour;

FinSommeVecteur



Algorithme

Algorithme Somme Vecteur
Entrée: U,V deux vecteurs de Z^n
Sortie: W un vecteur de Z^n
Local: i ∈ N

début
pour i ← 1 **jusquà** n **faire**
lire(Ui , Vi)
Fpour;
pour i ← 1 **jusquà** n **faire**
Wi ← Ui + Vi
Fpour;
pour i ←1 **jusquà** n **faire**
ecrire(Wi)
Fpour;
fin //sommevecteur

Delphi

```
program somme_vecteur;
const   n =3;
type
   intervalle=1..n;
   vecteur = array[intervalle] of integer;
var
   U,V,W:vecteur;
   i:intervalle;
begin
   for i:=1 to n do   readln (U[i],V[i]);
   for i:=1 to n do   W[i]:=U[i]+V[i];
   for i:=1 to n do   writeln(W[i]);
end.
```

Fonctions booléennes

Enoncé :

Programme simple d'écriture de deux fonctions de calcul des deux connecteurs logiques, le **et** booléen et le **ou** booléen par application des propriétés suivantes :

X **et** Faux = Faux X **et** Vrai = X	X **ou** Vrai = Vrai X **ou** Faux = X

Solution en Ldfa avec les traductions de chaque fonction en Delphi-Pascal et en Java

LDFA	LDFA
Algorithme LeEt // un Et logique **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens **debut** **si** x = **Faux alors** **resultat** ← **Faux** **sinon** **resultat** ← y **fsi** **fin** // LeEt	**Algorithme** LeOu // un Ou logique **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens **debut** **si** x =**Vrai alors** **resultat** ← **Vrai** **sinon** **resultat** ← y **fsi** **fin** // LeOu

DELPHI-Pascal
 	
function Et(x,y : Boolean):Boolean;
begin
if x=faux **then**
result := false
else
result := y
end;
	DELPHI-Pascal
	
	function Ou(x,y : Boolean):Boolean;
	begin
	if x=true **then**
	result := true
	else
	result := y
	end;

Java boolean Et **(** boolean x , boolean y **)** **{** **if (** x == false **)** **return** false **;** **else** **return** y **;** **}**	**Java** boolean Ou **(** boolean x , boolean y **)** **{** **if (** x == true **)** **return** true **;** **else** **return** y **;** **}**

On pourra utiliser les raccourcis d'écriture suivants pour un algorithme-fonction :

Algorithme LeEt **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens ↓	**Algorithme** LeOu **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens ↓
Fonction LeEt (x , y: Booléens) : **resultat** Booléens	**Fonction** LeOu (x , y: Booléens) : **resultat** Booléens

Variantes sur la factorielle

Enoncé : Algorithme de calcul de la factorielle d'un entier **n! =n.(n-1)!** en utilisant différentes instructions de boucles.

Solution en Ldfa

par boucle tantque..ftant

```
Algorithme Factor
  Entrée:  n  ∈ Entier*
  Sortie: fact ∈ Entier*
  Local: i  ∈ N

 début
  lire(n) ;
  fact ← 1;
  i ←  2 ;
  Tantque i <= n Faire
   fact ← fact * i ;
   i ← i + 1 ;
  Ftant;
  Ecrire (n ,! =  , fact );
 Fin // Factor
```

par boucle pour…jusquà

```
Algorithme Factor
    Entrée: n  ∈ Entier*
    Sortie:      fact ∈ Entier*
    Local:       i  ∈ N

 début
  lire(n) ;
  fact ← 1;
  Pour i ← 2 jusquà n Faire
   fact ← fact*i ;
  Fpour;
  Ecrire (n ,! =  , fact );
 Fin // Factor
```

par boucle repeter.... jusqua

```
Algorithme Factor
Entrée:  n  ∈ Entier*
Sortie: fact ∈ Entier*
Local: i  ∈ Entier
 début
   lire(n) ;
   fact ← 1;
   i ←  2 ;
   Repeter
      fact ← fact * i ;
      i ← i + 1 ;
   jusquà i > n ;
   ecrire(n ,! =  , fact );
 fin // Factor
```

par récursivité à partir de la formule : n! =n.(n-1)!

```
Algorithme fact_recur

utilise   Fonction fact
 debut
  ecrire ('5! =',fact(5))
 fin // fact_recur

Fonction fact (n:entier) : resultat entier
debut
    si n = 0 alors    resultat ← 1
    sinon  resultat ← fact (n - 1) * n
    fsi
fin // fact
```

```
program Factor;
 var n , fact , i :integer ;
 begin
    readln(n);
    fact:=1;
    i:=2;
    repeat
       fact:=fact*i;
       i:=i+1
    until i > n ;
    writeln(n ,'! =  ', fact)
 end.
```

```
program Factor;
var n:integer;

 function fact  (n : integer ) : integer ;
  begin
     if n=0 then   result:=1
     else   result:=fact(n-1)*n
  end;// fact

begin
 readln(n);
 writeln(n ,'! =  ', fact(n));
end.
```

PGCD , PPCM de deux entiers

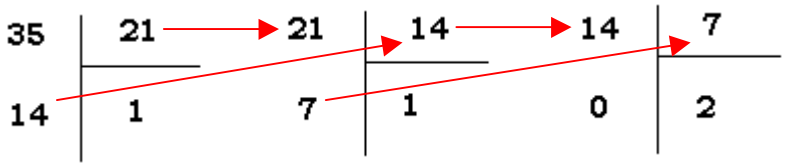
Enoncé : Ecrire des programmes LDFA donnant le pgcd et le ppcm de 2 entiers naturels.

> Le pgcd de 2 entiers non nuls est le plus grand diviseur commun à ces deux entiers.
> Exemple : 35 et 21 ont pour pgcd 7, car 35 = 7*5 et 21 = 7*3,

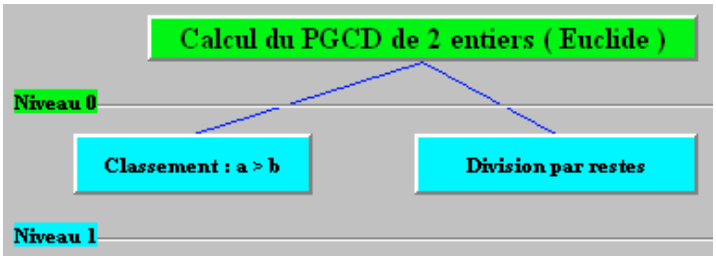
> Le ppcm de 2 entiers non nuls est le plus petit commun multiple à ces deux entiers.
> Exemple : 35 et 21 ont pour ppcm 105, car 105 =3*35 et 105=5*21.

Solution du pgcd faisant apparaître les niveaux de décomposition et l'algorithme associé

La méthode employée pour évaluer le pgcd de 2 entiers, se dénomme l'algorithme d'Euclide ou encore la division par les restes successifs. Soit le calcul du pgcd de 35 et 21 : on divise 35 par 21, puis 21 par le reste 14, puis 14 par le nouveau reste 7etc…Le processus s'arrête lorsque le dernier reste est nul, le pgcd est alors le précédent reste non nul



Le dernier reste non nul étant 7, le pgcd de 35 et 21 est donc 7. D'une manière générale, pour calculer le pgcd de deux entiers a et b nous divisons le plus grand par le plus petit, chacun de ces deux entiers est représenté par une variable a $\in$ **N** et b $\in$ **N**, nous classons systématiquement les contenus de ces variables en mettant dans **a** le plus grand des deux entiers et dans **b** le plus petit des deux.



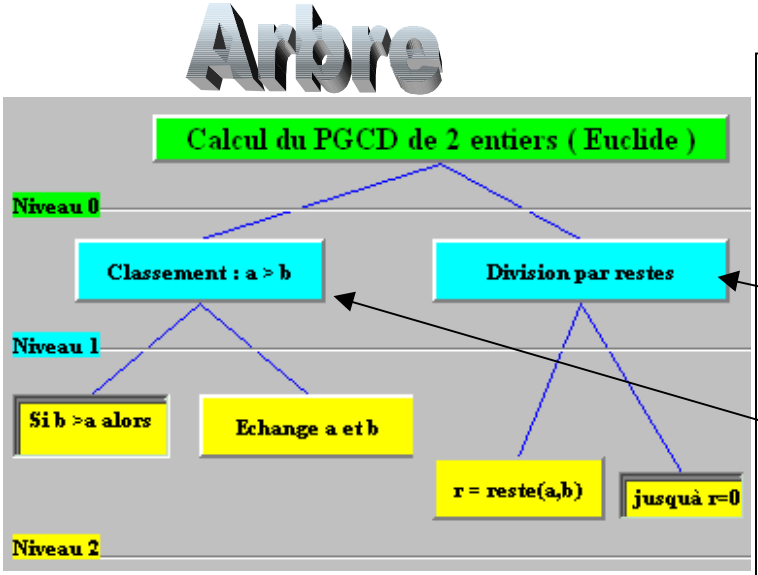
Deux actions sont utilisées pour calculer le pgcd de 2 entiers.

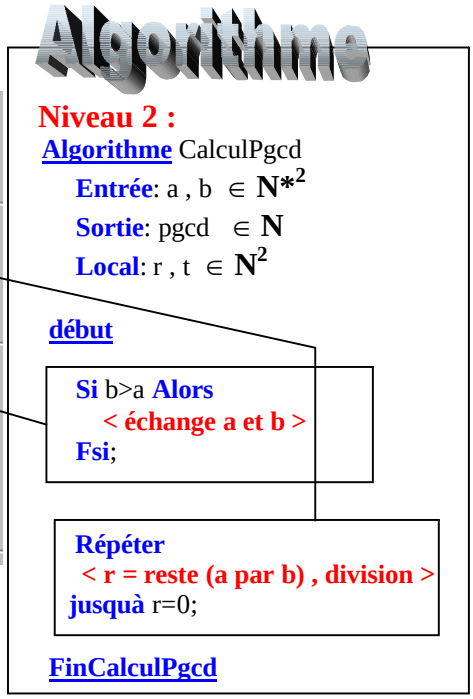
Niveau 1:

Algorithme CalculPgcd
Entrée: a , b $\in$ $\mathbf{N}^{*2}$
Sortie: pgcd $\in$ **N**
Local: r , t $\in$ $\mathbf{N}^2$

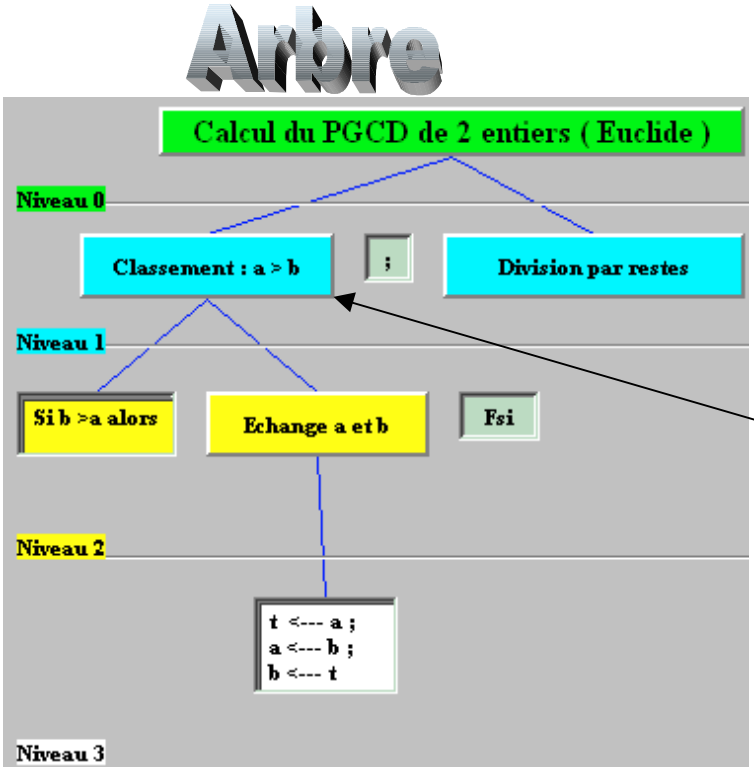
début
< min(a,b) dans b, max dans a >;
< divisions par restes successifs >
FinCalculPgcd

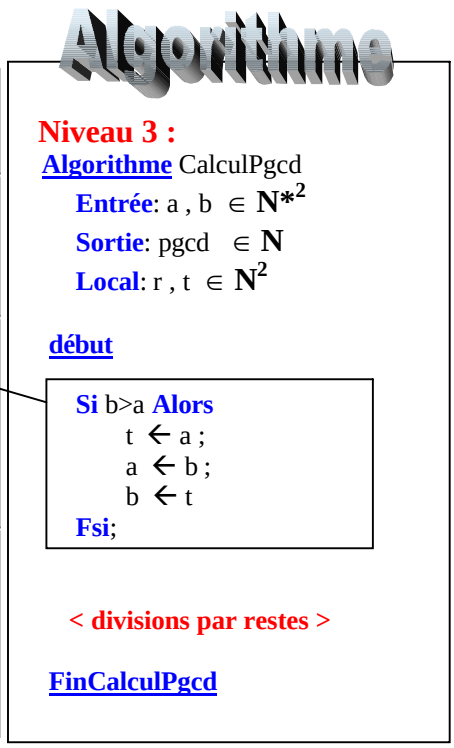
Action **< min(a,b) dans b, max dans a >:**	Action **< divisions par restes successifs >:**
Description :	*Description* :
Si b>a **Alors** **< échange a et b >** **Fsi**;	**Répéter** **< r = reste (a par b) , division >** **jusquà** r=0; pgcd ← a;



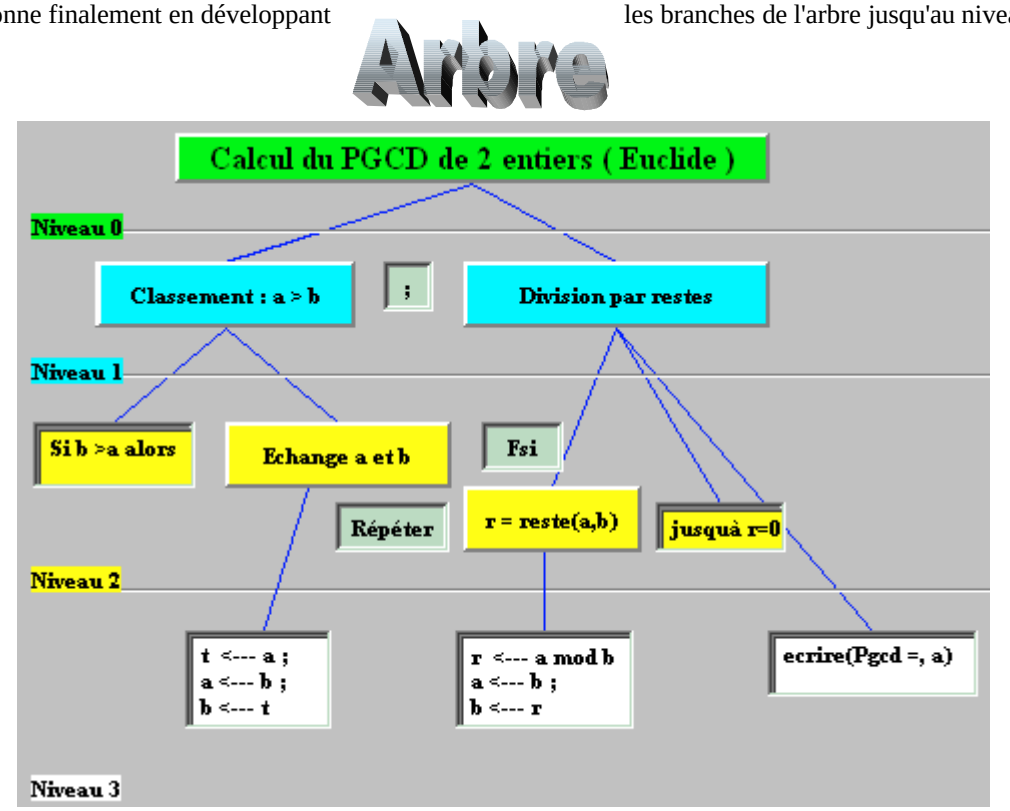


Action < **échange a et b** >	Action < **r = reste (a par b) , division** >
Description :	*Description* :
t ← a ; a ← b ; b ← t	r ← a mod b ; a ← b ; b ← r





Ce qui donne finalement en développant les branches de l'arbre jusqu'au niveau 3 :



Algorithme	Delphi

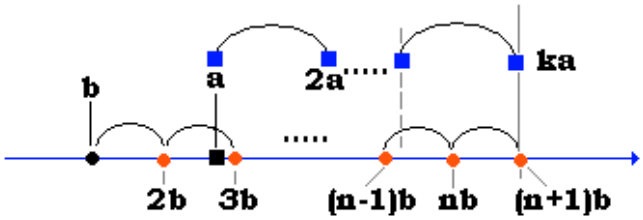
Algorithme CalculPgcd
Entrée: a ,b $\in$ N^{*2}
Sortie: pgcd $\in$ N
Local: r , t $\in$ N^2
 	
début
lire(a,b);
Si b>a **Alors**
t ← a ;
a ← b ;
b ← t
Fsi;
Répéter
r ← a mod b ;
a ← b ;
b ← r
jusquà r=0;
pgcd ← a;
ecrire(pgcd)
FinCalculPgcd
	program calcul_Pgcd;
	var
	a , b : integer;
	pgcd, r, t : integer;
	
	begin
	readln(a,b);
	if b>a **then**
	begin
	t := a ;
	a := b ;
	b := t
	end;
	repeat
	r := a **mod** b ;
	a := b ;
	b := r
	until r=0;
	pgcd:= a;
	writeln(pgcd)
	end.

Solution du calcul du ppcm de deux entiers sous forme d'un algorithme-fonction

La méthode employée pour évaluer le ppcm de 2 entiers a et b (b <a)est simple :

Nous évaluons tous les multiples de b par ordre croissant (2b, 3b, …) et nous arrêtons le calcul dès qu'un multiple est divisible par a (si a et b sont premiers entre eux le ppcm est égal à leur produit) :

Algorithme




```
Fonction ppcm (a,b:entier) résultat entier;
local : n , p ∈ entier

 debut
   p ← b;
   n ← 1;
   tantque (n <= a) et (p Mod a<>0) faire
        p ← b * n;
        n ← n + 1;
   ftant
   résultat ← p
 Fin // ppcm
```

Delphi

```
program  calcul_ppcm;
 var
  a, b, p:integer;

 function ppcm (a,b:integer):integer;
 var
  k,p:integer;
  begin
     p:=b;
     k:=1;
     while (k<= a)and(p mod a<>0) do
     begin
        p:= b * k;
        k:= k + 1;
     end;
     result:=p
  end;

  begin
     a:= 125;
     b:= 45;
     p:= ppcm(a, b);
     writeln('Calcul du ppcm :');
     writeln('a=', a,' b=', b,' => ppcm=', p)
  end.
```

Java

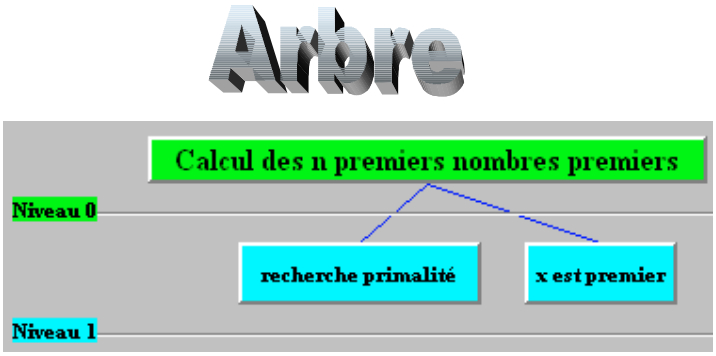
```
class ApplicationPpcm {
 public static void main(String[ ] args)  {
  int a,b,p;
  a=125;
  b=45;
  p=ppcm(a,b);
  System.out.println("Calcul du ppcm :");
  System.out.println("a="+a+" b="+b+" => ppcm="+p);
 }

    static int ppcm (int a , int b)
    {
        int n=1 , p=b;
        while((n <= a)&(p % a !=0))
        {
            p= b * n;
            n ++;
        }
        return p ;
    }
 /* Autre version avec un for sans aucun corps. Le code est
    plus compact, mais il est moins lisible !
 */
    static int ppcm (int a , int b)
    {
        int p=b;
        for(int n=1; (n<=a)&(p%a!=0); n++,p=b*n);
         return p ;
    }

}
```

Nombres premiers

Enoncé : Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même.
Exemple : 17 , 19, 31 sont des nombres premiers.

Ecrire un programme LDFA donnant les **n** premiers nombres premiers.

Solution faisant apparaître les niveaux de décomposition et l'algorithme associé



Deux actions sont utilisées pour calculer les nombres premiers, elles correspondent chacune à une branche de l'arbre.

Algorithme

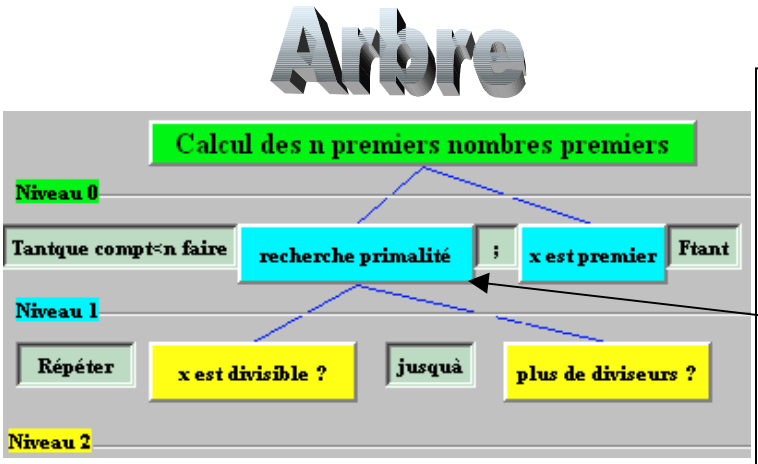
Niveau 1:

Algorithme Premier
Entrée: n $\in$ **N**
Sortie: x $\in$ **N**
Local: Est_premier $\in$ {**Vrai** , **Faux**}
Divis , compt $\in$ $\mathbf{N}^2$

début
lire(n);
Tantque(compt < n) **Faire**
< recherche primalité de x >;
< comptage si x est premier >
Ftant
FinPremier

Action **< recherche primalité de x >**	Action **< comptage si x est premier >**
Description :	*Description* :
Répéter **< x est-il divisible ?>** **jusquà < non premier, ou plus de diviseurs>**	**Si** Est_premier =**Vrai** **Alors** **<on ecrit x et on le compte>** **Fsi**; **<on passe au x suivant>**

Etude de la branche gauche de l'arbre au niveau 2 :



Algorithme

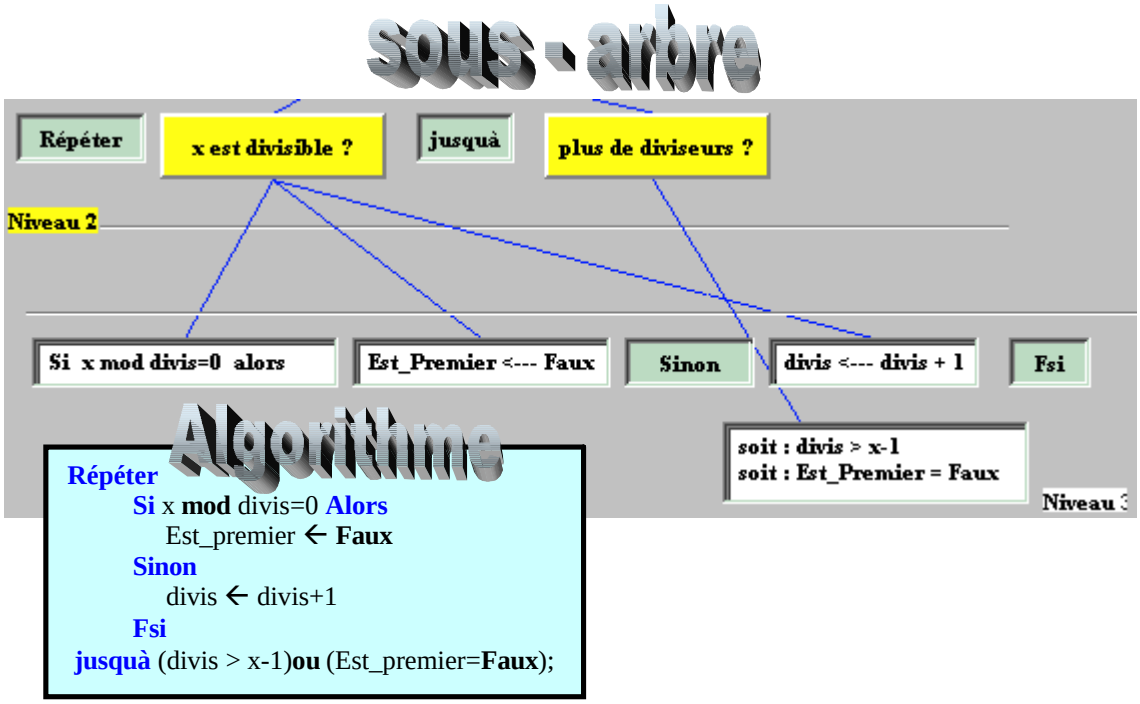
Niveau 2 :
début
lire(n);
Tantque (compt < n) **Faire**

Répéter
< x est-il divisible ?>
jusquà < plus de diviseurs>

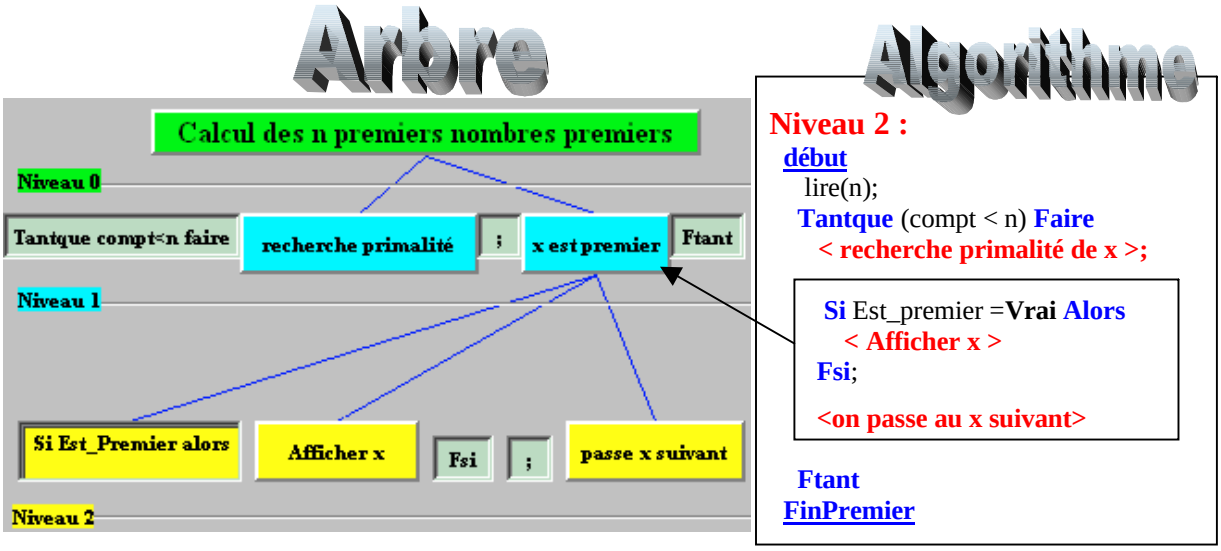
< comptage si x est premier >
Ftant
FinPremier

Etude de la branche gauche de l'arbre au niveau 3 :

Action < x est-il divisible ?>	Action < non premier, ou plus de diviseurs>
Description : **Si** x mod divis=0 **Alors** Est_premier ← **Faux** **Sinon** divis ← divis+1 **Fsi**	*Description* : (divis > n-1) **ou** (Est_premier=**Faux**)

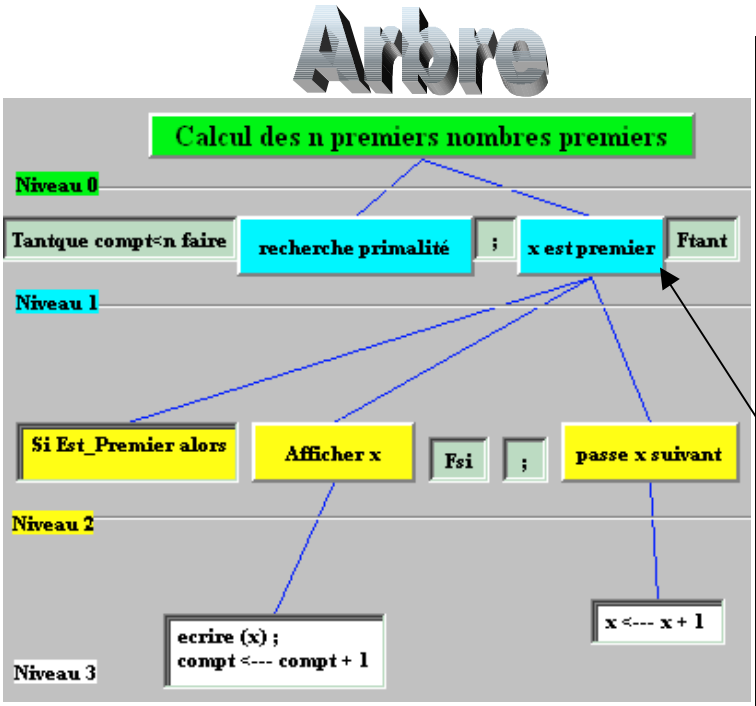


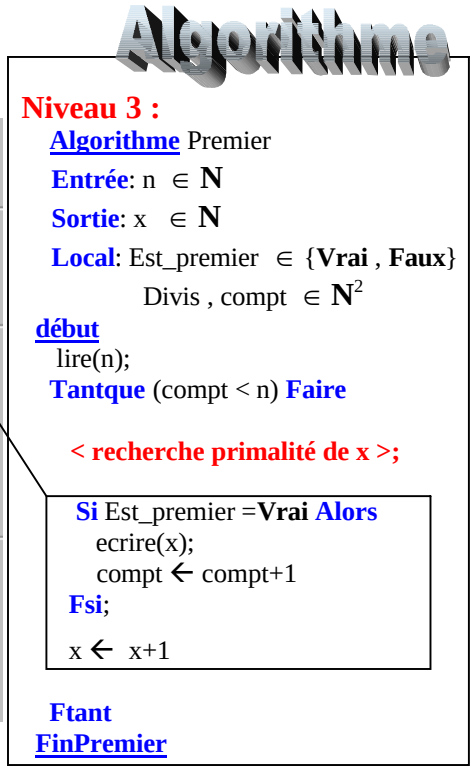
Etude de la branche droite de l'arbre au niveau 2 :



Etude de la branche droite de l'arbre au niveau 3 :

Action < **Afficher x** >	Action **<on passe au x suivant>**
Description : ecrire(x); compt ← compt+1	*Description :* x ← x+1





Version finale complète de l'algorithme → **sa traduction en Delphi**

```
Algorithme Premier
  Entrée: n ∈ N
  Sortie: x ∈ N
  Local: Est_premier ∈ {Vrai , Faux}
         Divis , compt ∈ N²
début
 lire(n);
 compt ← 1;
 ecrire(2);
 x ← 3;
 Tantque(compt < n) Faire
  divis ← 2;
  Est_premier ← Vrai;
  Répéter
    Si x mod divis=0 Alors     Est_premier ← Faux
    Sinon     divis ← divis+1
    Fsi
  jusquà (divis > x-1)ou (Est_premier=Faux);
  Si Est_premier =Vrai Alors
    ecrire(x);
    compt ← compt+1
  Fsi;
  x ← x+1
 Ftant
FinPremier
```

Algorithme

```
program Premier;
 var
  n,nbr,divis,compt:integer;
  Est_premier:boolean;
  begin
   write('Combien de nombres premiers : ');
   readln(n);
   compt:=1;
   writeln(2);
   nbr:= 3;
   while (compt < n) do
   begin
    divis:= 2;
    Est_premier:= true;
    repeat
      if nbr mod divis=0 then    Est_premier:= false
      else     divis:= divis+1
    until   (divis > nbr div 2)or (est_premier=false);
    if Est_premier=true then
    begin
     writeln(nbr);
     compt:= compt+1
    end;
    nbr:= nbr+1
   end
  end.
```

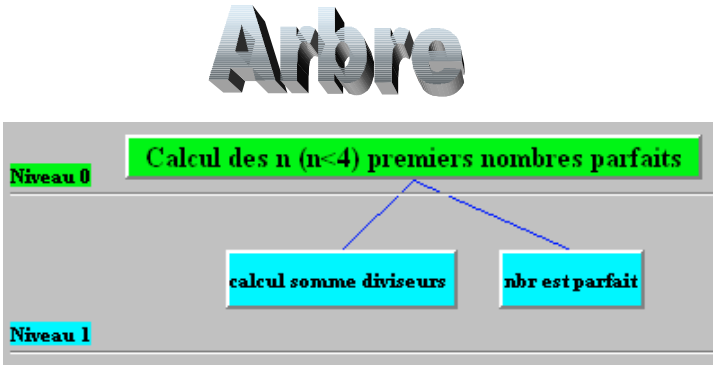
Delphi

Nombres parfaits

Enoncé : Un nombre est dit parfait s'il est égal à la somme de ses diviseurs 1 compris.
Exemple : 6 = 1+2+3 , est parfait

Ecrire un programme LDFA donnant les **n** premiers nombres parfaits.

Solution faisant apparaître les niveaux de décomposition et l'algorithme associé



Deux actions sont utilisées pour calculer les nombres parfaits, elles correspondent chacune à une branche de l'arbre.

Algorithme

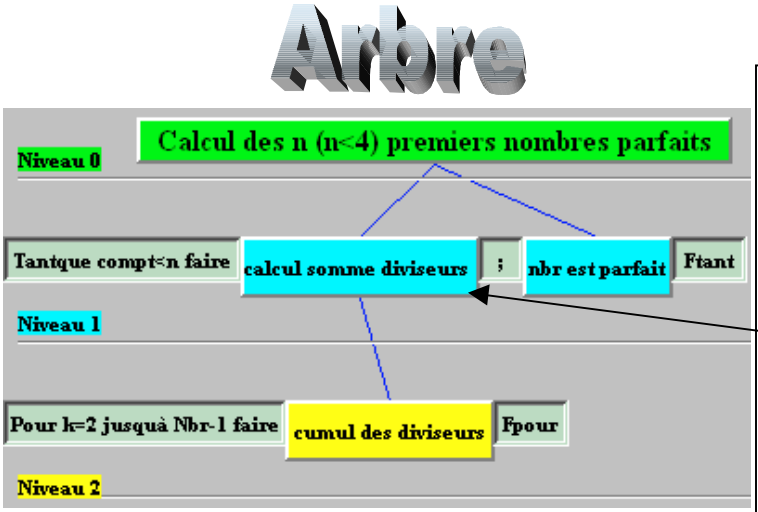
Niveau 1:

Algorithme Parfait
Entrée: n $\in$ **N**
Sortie: nbr $\in$ **N**
Local: som, k, compt $\in$ $\mathbf{N}^3$

début
Tantque (compt < n) **Faire**
< somme des diviseurs de nbr >;
< nbr est parfait >
Ftant
FinParfait

Action **< somme des diviseurs de nbr >**	Action **< nbr est parfait >**
Description : *calcul de la somme des diviseurs du nombre : **nbr*** **Pour** k ← 2 **jusquà** nbr-1 **Faire** **< cumul, si k divise nbr >** **Fpour**	*Description :* *lorsque le nombre « **nbr** » est reconnu comme parfait, il doit être compté, puis affiché à l'écran* **< nbr est parfait si nbr = som >** **< comptage >**

Etude de la branche gauche de l'arbre au niveau 2 :



Algorithme

Niveau 2 :

Début

Tantque (compt < n) **Faire**

pour k ← 2 **jusquà** nbr-1 **Faire**
< cumul des diviseurs >
Fpour;

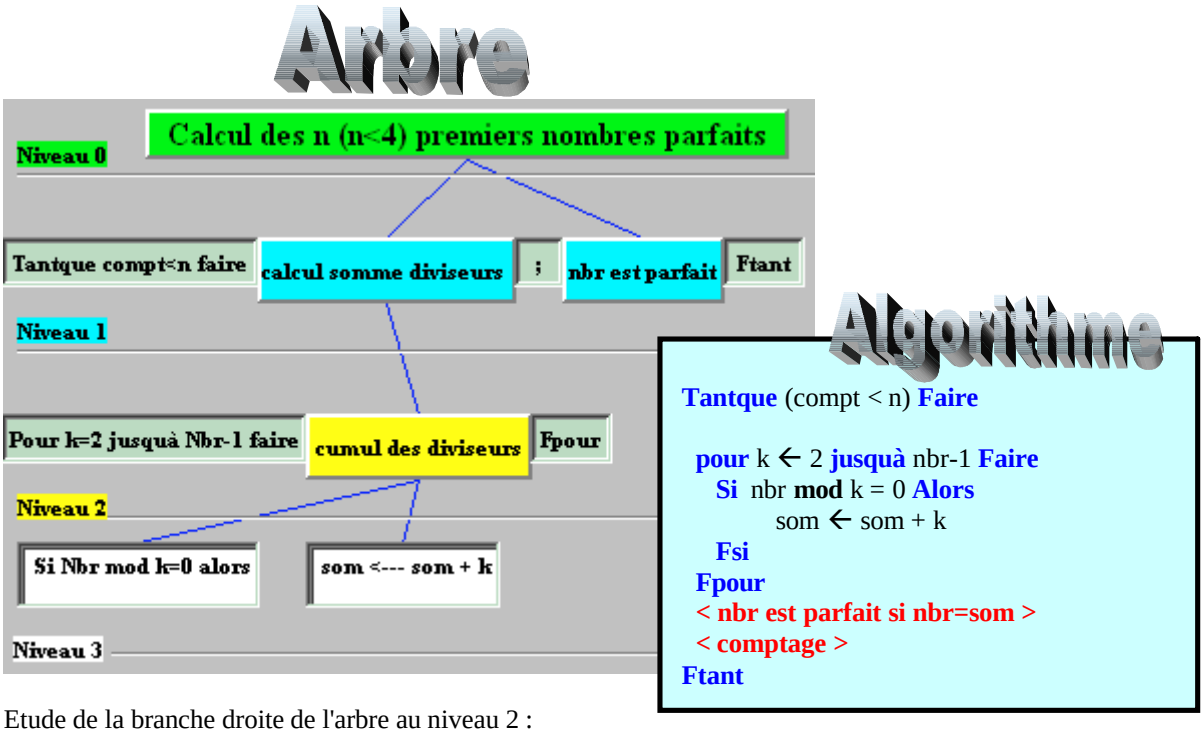
< nbr est parfait si nbr=som >
< comptage >
Ftant
FinParfait

Etude de la branche gauche de l'arbre au niveau 3 :

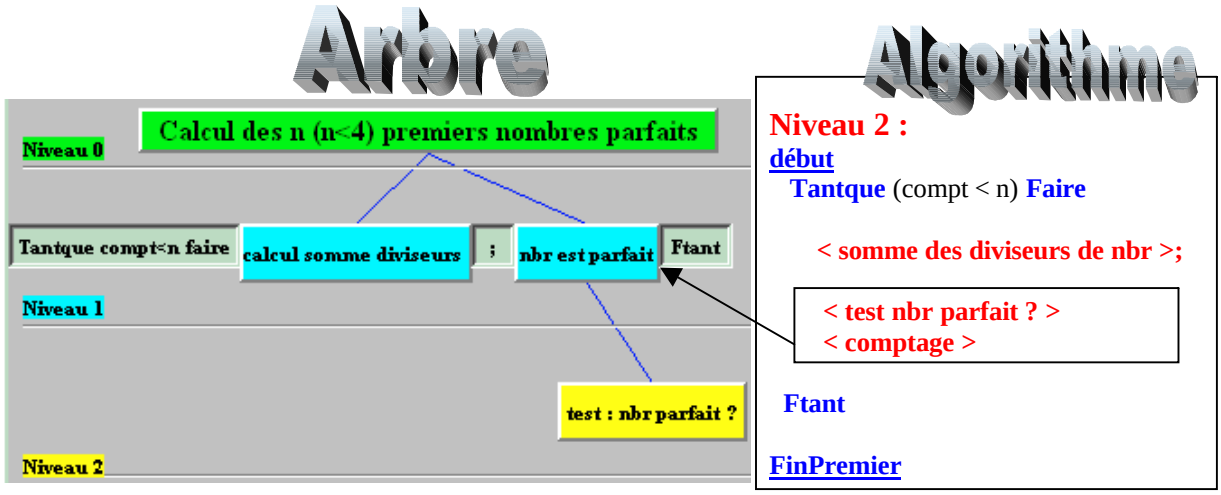
Action < **cumul des diviseurs, (si k divise nbr)** >

*on cumule **k** dans la variable **som** (somme des diviseurs) du nombre **nbr** lorsque **k** divise effectivement **nbr**.*

Si nbr **mod** k = 0 **Alors**
    som ← som + k
Fsi

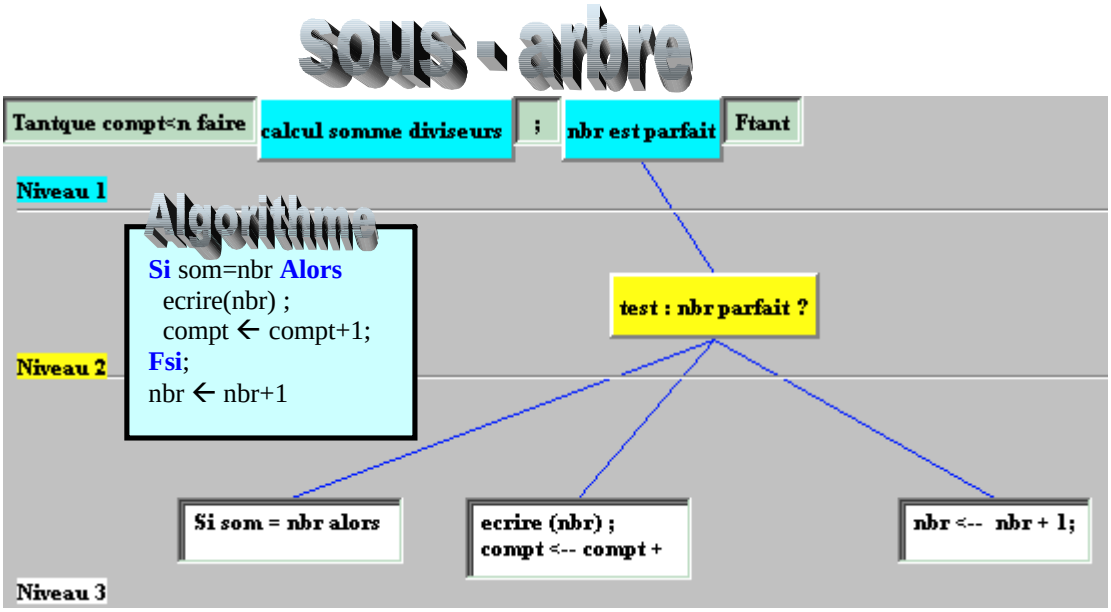


Etude de la branche droite de l'arbre au niveau 2 :



Action < **test nbr parfait ?** >	Action < **comptage** >
Description : *le nombre **nbr** est parfait s'il est égal à la somme calculée :* **Si** som=nbr **Alors** ecrire(nbr) ; compt ← compt+1; **Fsi**;	*Description :* *Puis l'on passe au nombre suivant* nbr ← nbr+1

Etude de la branche droite de l'arbre au niveau 3 :



Version finale complète de l'algorithme → **sa traduction en Delphi**

Algorithme Parfait
  Entrée: n ∈ **N**
  Sortie: nbr ∈ **N**
  Local: som, k, compt ∈ $\mathbf{N}^3$

début
 lire(n);
 compt ← 0;
 nbr ← 2;
 Tantque (compt < n) **Faire**
   som ← 1;
   Pour k ← 2 **jusquà** nbr-1 **Faire**
     Si nbr mod k = 0 **Alors**
       som ← som + k
     Fsi
   Fpour ;
   Si som=nbr **Alors**
     ecrire(nbr) ;
     compt ← compt+1;
   Fsi;
   nbr ← nbr+1
 Ftant

FinParfait

```
program Parfait;
 var
  n, nbr : integer;
  som, k, compt : integer;

 begin
   readln(n);
   compt := 0;
   nbr := 2;
   while (compt < n) do
   begin
     som:= 1;
     for k:= 2 to nbr-1 do
      if nbr mod k=0 then
        som := som + k ;
     if som=nbr then
     begin
       writeln(nbr);
       compt:= compt+1;
     end ;
     nbr:= nbr+1
   end
 end.
```

Suite : racine carrée - Newton

Enoncé : Etude d'une suite convergente de la forme $U_n = f(U_{n-1})$

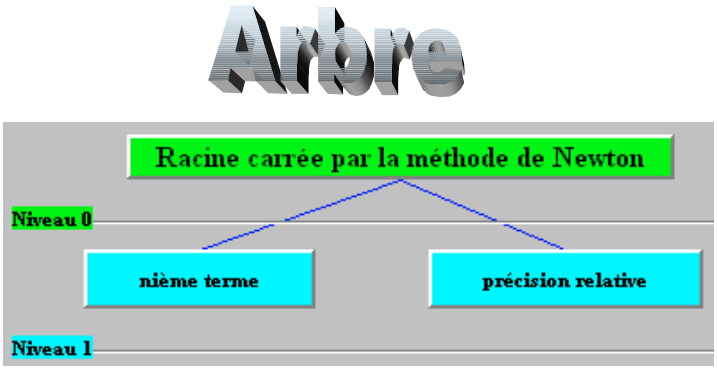
Ecrire un programme LDFA proposant une étude simple de la suite U_n suivante (méthode de Newton) :

$$U_n = 1/2(U_{n-1} + X/ U_{n-1}) \quad X>0$$

La suite U_n converge vers le nombre $\sqrt{X}$ (la racine carré de X), le programme doit effectuer :

1° le calcul du terme de rang n donné par l'utilisateur,
2° le calcul jusqu'à une précision relative Epsilon fixée

Solution faisant apparaître les niveaux de décomposition et l'algorithme associé



Deux actions sont utilisées pour effectuer les calculs demandés, elles correspondent chacune à une branche de l'arbre.

Algorithme

Niveau 1:

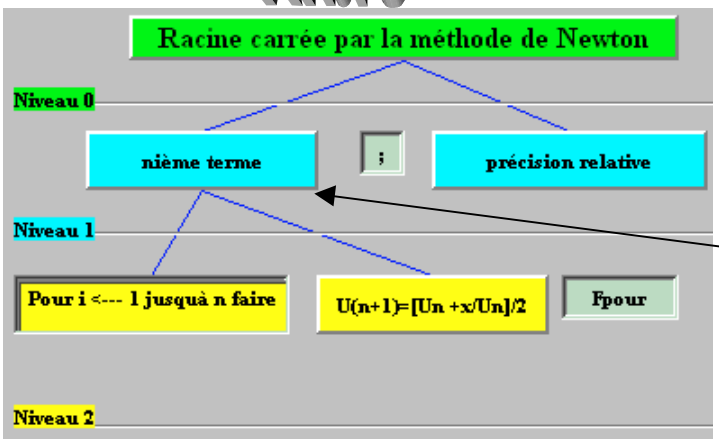
Algorithme Newton
Entrée: n $\in$ **N***
x $\in$ **R***
$\varepsilon \in$ **R** ($\varepsilon \in [0,1]$)
Sortie: u $\in$ **R**
Local: v $\in$ **R**
i $\in$ **N**
début

<calcul du terme de rang n>;
<calcul de la limite à la précision ε **>**

FinNewton

Action **<calcul du terme de rang n>**	Action **<calcul de la limite à la précision** ε **>**
Description : **Pour** i ← 1 **jusquà** n **Faire** **< u(n+1) ← [u(n)+x/u(n)]/2 >** **Fpour**;	*Description* : **Répéter** **<* u(n+1)=[u(n)+x/u(n)]/2 , précision *>** **jusquà <* précison <** ε ***>**

Etude de la branche **Arbre** gauche de l'arbre au niveau 2 : **Algorithme**



Niveau 2:

début
Eps ← 10^{-4};
n ←10;
lire(x);
Pour i ← 1 **jusquà** n **Faire**
< u(n+1) ← [u(n)+x/u(n)]/2 >
Fpour;
ecrire(u);

<calcul de la limite à la précision ε **>**
FinNewton

Etude de la branche gauche de l'arbre au niveau 3 :

Action < u(n+1) ← [u(n)+x/u(n)]/2 >

v ← u;
u ← (u + x/u)/2 ;

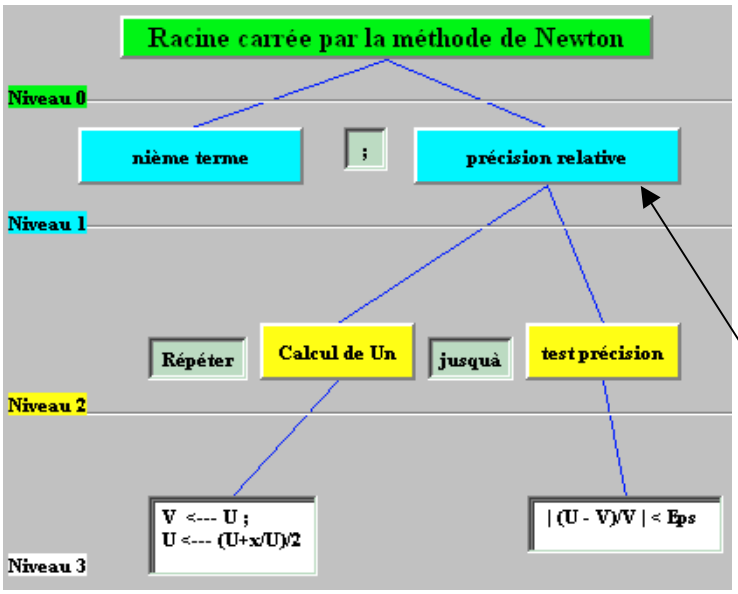
Algorithme

Niveau 3:

début
Eps ← 10^{-4};
n ←10;
lire(x);
Pour i ← 1 **jusquà** n **Faire**
v ← u;
u ← (u + x/u)/2 ;
Fpour;
ecrire(u);
<calcul de la limite à la précision ε >
FinNewton

Développement de la branche droite de l'arbre jusqu'au niveau 3 :

Arbre



Algorithme

Niveau 3:

début
Eps ← 10^{-4};
n ←10;
lire(x);
Pour i ← 1 **jusquà** n **Faire**
v ← u;
u ← (u + x/u)/2 ;
Fpour;
ecrire(u);
Répéter
v ← u;
u ← (u + x/u)/2 ;
jusquà | (u-v)/v | < Eps;

ecrire(u)

FinNewton

Version finale complète de l'algorithme → **sa traduction en Delphi**

Algorithme Newton
Entrée:
n ∈ **N***, x ∈ **R***, ε ∈ **R** (ε ∈ [0,1])
Sortie: u ∈ **R**
Local: v ∈ **R**, i ∈ **N**
 	
début
Eps ← 10^{-4};
n ←10;
lire(x);
// calcul du terme de rang n donné:
u ← (1 + x)/2 ;
Pour i ← 1 **jusquà** n **Faire**
u ← (u + x/u)/2 ;
Fpour;
ecrire(u);
	program Newton;
	
	const
	Eps=1E-4;
	n=10;
	var
	u,v,x : real ;
	i : integer ;
	
	begin
	readln(x);
	// calcul du terme de rang n donné:
	u:=(1+x)/2;
	for i:= 1 **to** n **do**
	u:= (u + x/u)/2 ;
	writeln(u);

Algorithme

Delphi

Algorithme

suite

```
//calcul jusqu'à une précision Epsilon fixée

u ← (1 + x)/2 ; { réinitialisation }
Répéter
 v ← u;
 u ← (u + x/u)/2 ;
jusquà | (u-v)/v | < Eps;
ecrire(u)

FinNewton
```

Delphi

```
//calcul jusqu'à une précision Epsilon fixée

u:=(1+x)/2;
 repeat
   v:= u;
   u:= (u+x/u)/2
 until   abs((u-v)/v ) < eps;
 writeln(u)

 end.
```

Java

```
import Readln;

class Newton
{
 public static void main (String [ ] arg)  {

  final double Eps=1E-4;
  int n=10;
  double u,v,x;
  x = Readln.undouble();
  // calcul du terme de rang n donné:
  u = (1+x)/2;
  for(int i=1;i<=n;i++)
   u = (u + x/u)/2 ;
  System.out.println("1°) après "+n+" termes: "+u);

   //calcul jusqu'à une précision Epsilon fixée:
  u=(1+x)/2;
  do
   {
    v = u;
    u = (u+x/u)/2;
   }
  while(Math.abs((u-v)/v ) >= Eps);
  System.out.println("2°) à la précision "+Eps+": "+u);
 }
}
```

Inversion d'un tableau

Enoncé : Autre programme simple d'utilisation des tableaux, écrire un programme LDFA inversant le contenu d'un tableau à n éléments entiers déjà rempli, on écrira le contenu du tableau avant inversion, puis son contenu après inversion.

Solution en Ldfa avec les traductions de chaque fonction en Delphi-Pascal et en Java

Algorithme

```
Algorithme InverseTable
  Global: Table; vecteur de N^n
  Local: temp ∈ N , i ∈ N
         ( i ∈ [1 , Max] )
début
{remplissage aléatoire du tableau}

 Pour i ← 1 jusquà Max Faire
   ecrire (Table_i )
 Fpour;
 Pour i ← 1 jusquà Ent[Max/2] Faire
      Temp ← Table_i ;
      Table_i ← Table_(Max - i + 1) ;
      Table_(Max - i + 1) ← Temp
 Fpour;
 Pour i ← 1 jusquà Max Faire
    ecrire (Table_i )
 Fpour;

FinInverseTable
Ent[p] représente la partie entière de p
```

Delphi

```
program inverse_tableau;
 const
   Max=10;
 type
   intervalle=1..Max;
   Tableau=array[intervalle] of integer;
 var
   Table:Tableau;
   i:intervalle;
   Temp:integer;
  begin
  {remplissage aléatoire du tableau}
  for i:=1 to Max do
   Table[i]:=random(1000);

  {voir le contenu du tableau  avant opération :  }
  for i:=1 to Max do
   writeln('i=',i:2,':',Table[i]:4);

  for i:=1 to Max div 2 do
  begin
   Temp:=Table[i];
   Table[i]:=Table[Max-i+1];
   Table[Max-i+1]:=Temp
  end;
  writeln('-----------------');
  {voir le contenu du tableau  après opération :  }
  for i:=1 to Max do
   writeln('i=',i:2,':',Table[i]:4);
 end.
```

```
class InvTab{
 public static void main (String [ ] arg)   {
  final int Max=6;
  long[ ]table= new long[Max+1];
  //remplissage aléatoire du tableau
  for(int i=0;i<=Max;i++)
   table[i] = Math.round(Math.random( )*100);

  //voir le contenu du tableau avant opération
  for(int i=0;i<=Max;i++)
   System.out.println("table["+i+"] =  "+
table[i]);

   for(int i=0;i<=Max/2;i++)  {
     long Temp=table[i];
     table[i]=table[Max-i];
    table[Max-i]=Temp;
   }
 }
}
```

Chapitre 4 : Structures de données

4.1.Spécification abstraite de données

- Le Type Abstrait Algébrique (TAA)
- Disposition pratique d'un TAA
- Le Type Abstrait de Donnée (TAD)
- Classification hiérarchique
  - le TAD liste linéaire
  - le TAD pile LIFO
  - le TAD file FIFO
- Exercices d'implantation de TAD

4.2. Implantation de TAD avec Unit en Delphi

- **Types abstraits de données et Unités en Delphi**

  Traduction générale TAD → Unit pascal
  Exemples de Traduction TAD → Unit pascal
  Variations sur les spécifications d'implantation
  Exemples d'implantation de la liste linéaire
  Exemples d'implantation de la pile LIFO
  Exemples d'implantation de la file FIFO

4.3. Structures d'arbres binaires

- Vocabulaire employé sur les arbres :
  - Etiquette Racine, noeud, branche, feuille
  - Hauteur, profondeur ou niveau d'un noeud
  - Chemin d'un noeud , Noeuds frères, parents, enfants, ancêtres
  - Degré d'un noeud
  - Hauteur ou profondeur d'un arbre
  - Degré d'un arbre
  - Taille d'un arbre

- Exemples et implémentation d'arbre
  - Arbre de dérivation
  - Arbre abstrait
  - Arbre lexicographique
  - Arbre d'héritage
  - Arbre de recherche

- Arbres binaires
- TAD d'arbre binaire
- Exemples et implémentation d'arbre
  - *tableau statique*
  - *variable dynamique*
  - *classe*
- Arbres binaires de recherche
- Arbres binaires partiellement ordonnés *(tas)*
- Parcours en largeur et profondeur d'un arbre binaire
  - Parcours d'un arbre
  - Parcours en largeur
  - Parcours préfixé
  - Parcours postfixé
  - Parcours infixé
  - Illustration d'un parcours en profondeur complet

- Exercices deux TAD implantés en Unit et en classes avec Delphi

4.1 : Spécification abstraite de donnée

Plan du chapitre:

Introduction

Notion d'abstraction
spécification abstraite
spécification opérationnelle

1. La notion de TAD (type abstrait de données)

1.1 Le Type Abstrait Algébrique (TAA)
1.2 Disposition pratique d'un TAA
1.3 Le Type Abstrait de Donnée (TAD)
1.4 Classification hiérarchique
1.5 Le TAD liste linéaire (spécifications abstraite et concrète)
1.6 Le TAD pile LIFO (spécification abstraite et concrète)
1.7 Le TAD file FIFO (spécification abstraite seule)

Exercices d'implantation de TAD

Introduction

Le mécanisme de l'abstraction est l'un des **plus** important avec celui de la méthode structurée fonctionnelle en vue de la réalisation des programmes.

Notion d'abstraction en informatique

L'abstraction consiste à penser à un objet en termes d'actions que l'on peut effectuer sur lui, et non pas en termes de représentation et d'implantation de cet objet.

C'est cette attitude qui a conduit notre société aux grandes réalisations modernes. C'est un art de l'ingénieur et l'informatique est une science de l'ingénieur.

Dès le début de la programmation nous avons vu apparaître dans les langages de programmation les notions de **subroutine** puis de **procédure** et de **fonction** qui sont une abstraction d'encapsulation pour les familles d'instructions structurées:

- Les paramètres formels sont une **abstraction** fonctionnelle (comme des variables muettes en mathématiques),
- Les paramètres effectifs au moment de l'appel sont des **instanciations** de ces paramètres formels (une implantation particulière).

L'idée de considérer les **types** de données comme une abstraction date des années 80. On s'est en effet aperçu qu'il était nécessaire de s'abstraire de la représentation ainsi que pour l'abstraction fonctionnelle. On a vu apparaître depuis une vingtaine d'année un domaine de recherche : celui des **spécifications algébriques**. Cette recherche a donné naissance au concept de **Type Abstrait Algébrique** (TAA).

Selon ce point de vue une structure de donnée devient:

Une collection d'informations structurées et reliées entre elles selon un graphe relationnel établi grâce aux opérations effectuées sur ces données.

Nous spécifions d'une façon simple ces structures de données selon deux niveaux d'abstraction, du plus abstrait au plus concret.

Une spécification abstraite :

Description des propriétés générales et des opérations qui décrivent la structure de données.

Une spécification opérationnelle :

Description d'une forme d'implantation informatique choisie pour représenter et décrire la structure de donnée. Nous la divisons en deux étapes : la spécification opérationnelle concrète (choix d'une structure informatique classique) et la spécification opérationnelle d'implantation (choix de la description dans le langage de programmation).

Remarque :

Pour une spécification abstraite fixée nous pouvons définir plusieurs spécifications opérationnelles différentes.

1. La notion de TAD (Type Abstrait de Données)

Bien que nous situant au niveau débutant il nous est possible d'utiliser sans effort théorique et mental compliqué, une méthode de spécification semi-formalisée des données. Le " type abstrait de donnée " basé sur le type abstrait algébrique est une telle méthode.
Le lecteur ne souhaitant pas aborder le formalisme mathématique peut sans encombre pour la suite, sauter le paragraphe qui suit et ne retenir que le point de vue pratique de la syntaxe d'un TAA.

1.1 Le Type Abstrait Algébrique (TAA)

Dans ce paragraphe nous donnons quelques indications théoriques sur le support formel algébrique de la notion de TAA. *(notations de F.H.Raymond cf.Biblio)*

Notion d'algèbre formelle informatique

Soit (Fn) $n \in \mathbf{N}$ **,** une famille d'ensembles tels que :

$(\exists i_0 \in \mathbf{N}\ (1 \leq i_0 \leq n) / Fi_0 \neq \varnothing) \wedge (\forall i, \forall j,\ i \neq j \Rightarrow F_i \cap F_j = \varnothing)$
posons : $\mathbf{I} = \{ n \in \mathbf{N} / \mathbf{F}_n \neq \varnothing \}$

Vocabulaire :

Les symboles ou éléments de F_0 sont appelés symboles de constantes ou symboles fonctionnels 0-aire.

Les symboles de F_n (où $n \geq 1$ et $n \in \mathbf{I}$) sont appelés symboles fonctionnels n-aires.

Notation :

$$F = \bigcup_{n \in N} Fn = \bigcup_{n \in \mathbf{I}} Fn$$

soit F* l'ensemble des expressions sur F, **le couple (F^*,F) est appelé une algèbre abstraite**.

On définit pour tout symbole fonctionnel n-aire f, une application de F^*_n dans F^* notée "$\hat{f}$" de la façon suivante :

$\forall$ e, ($f \in F_n \rightarrow \hat{f}$)

et { $\hat{f}$: $F^{*n} \rightarrow F^*$ telle que $(a_1,..., a_n) \rightarrow \hat{f}$ (a1,...,an) = f(a1,...,an) }

On dénote :

$$F_n = \{ \hat{f} / f \in Fn \} \text{ et } \hat{F} = \bigcup_{n \in I} Fn$$

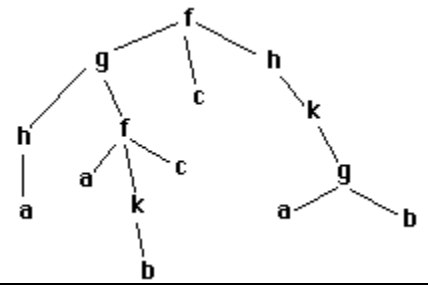
le couple $(F^*, \hat{F})$ est appelé une algèbre formelle informatique (AFI).

Les expressions de F^* construites à partir des fonctions $\hat{f}$ sur des symboles fonctionnels n-aires s'appellent des schémas fonctionnels.

Par définition, les schémas fonctionnels de F_0 sont appelés les constantes.

Exemple :
F0 = {a,b,c} // *les constantes*
F1 = {h,k} // *les symboles unaires*
F2 = {g} // *les symboles binaires*
F3 = {f} // *les symboles ternaires*

Soit le schéma fonctionnel : **fghafakbcchkgab**
(chemin aborescent abstrait non parenthésé)

Ce schéma peut aussi s'écrire avec un parenthèsage :	ou encore avec une représentation arborescente:
f [g(h(a),f(a,k(b),c)), c, h(k(g(a,b)))]	

Interprétation d'une algèbre formelle informatique

Soit une algèbre formelle informatique (AFI) : **(** $F^*, \hat{F}$ **)**

- on se donne un ensemble **X** tel que **X** $\neq \emptyset$,
- **X** est muni d'un ensemble d'opérations sur X noté Ω,

L'on construit une fonction ψ telle que :

ψ : **(** $F^*, \hat{F}$ **)** $\rightarrow$ **X** ayant les propriétés d'un homomorphisme

ψ est appelée fonction d'interprétation de l'AFI.
X est appelée l'univers de l'AFI.

Une AFI est alors un modèle abstrait pour toute une famille d'éléments fonctionnels, il suffit de changer le modèle d'interprétation pour implanter une structure de données spécifique.

Exemple :

$F_0 = \{x, y\}$ une AFI
$F_2 = \{f, g\}$
$F = F_0 \cup F_2$

l'Univers : **X** = **R** *(*les nombres réels)

les opérateurs $\Omega = \{+,*\}$ (addition et multiplication)

l'interprétation y : **(**$F^*, \hat{F}$**)** $\rightarrow$ (**R** **,** Ω)
définie comme suit :

ψ (f) : $\mathbf{R}^2 \rightarrow \mathbf{R}$
ψ (f) : (a,b) $\rightarrow$ ψ (f) [a,b] = a+b

ψ (g) : $\mathbf{R}^2 \rightarrow \mathbf{R}$
ψ (g) : (a,b) $\rightarrow$ ψ (g) [a,b] = a*b

ψ (x) = a_0 (avec $a_0 \in$ **R** fixé interprétant la constante x)
ψ (y) = a_1 (avec $a_1 \in$ **R** fixé interprétant la constante y)

Soit le schéma fonctionnel **fxgyx,** son interprétation dans ce cas est la suivante :

ψ(fxgyx) = ψ(f (x,g(y,x))) = ψ(f) **(**ψ(x),ψ(g)[ψ(y),ψ(x)] **)**
= ψ(x) + ψ(g)[ψ(y),ψ(x)] $\Leftarrow$ *propriété de ψ(f)*
= ψ(x) + ψ(y)*ψ(x) $\Leftarrow$ *propriété de ψ(g)*

Ce qui donne comme résultat : ψ (fxgyx) = $a_0 + a_1 * a_0$

A partir de la même AFI, il est possible de définir une autre fonction d'interprétation ψ'et un autre Univers **X**'.

par exemple :
l'Univers : **X = N** *(les entiers naturels)*
les opérateurs : Ω = **{ reste** , $\geq$ **}** *(le reste de la division euclidienne et la relation d'ordre)*

La fonction ψ'est définie comme suit :

ψ'(g) : $\mathbf{N}^2 \rightarrow \mathbf{N}$
ψ'(g) : (a,b) → ψ'(g)[a,b] = **reste**(a,b)

ψ'(f) : $\mathbf{N}^2 \rightarrow \mathbf{N}$
ψ'(f) : (a,b) → ψ'(f)[a,b] = a ≥ b ψ'(x) = n_0 (avec $n_0 \in$ **N** fixé)
ψ'(y) = n_1 (avec $n_1 \in$ **N fixé**)

On interprète alors le même schéma fonctionnel dans ce nouveau cas **fxgyx** :
ψ'(**fxgyx**) = n_0 ≥ **reste**(n_1, n_0)

Ceci n'est qu'une interprétation. cette interprétation reste encore une abstraction de plus bas niveau, le sens (sémantique d'exécution), s'il y en a un, sera donné lors de l'implantation de ces éléments. Nous allons définir un outil informatique se servant de ces notions d'AFI et d'interprétation, il s'agit du type abstrait algébrique.

Un TAA (type abstrait algébrique) est alors la donnée du triplet :

- une AFI
- un univers **X** et Ω
- une fonction d'interprétation ψ

la syntaxe du TAA est définie par l'**AFI** et l'ensemble **X**
la sémantique du TAA est définie par ψ et l'ensemble Ω

Notre objectif étant de rester pratique, nous arrêterons ici la description théorique des TAA (compléments cités dans la bibliographie pour le lecteur intéressé).

1.2 Disposition pratique d'un TAA

on écrira (exemple fictif):

Sorte : A, B, C ..... *les noms de types définis par le TAA, ce sont les types au sens des langages de programmation.*

Opérations :

f : A x B → B
g : A → C
x : → B *(notation pour les symboles de constantes de F0)*
y : → B *(notation pour les symboles de constantes de F0)*

Cette partie qui décrit la syntaxe du TAA s'appelle aussi **la signature du TAA** .

La sémantique est donnée par ψ , Ω sous la forme d'axiomes et de préconditions.

Le domaine d'une opération définie partiellement est défini par une précondition.

Un TAA réutilise des TAA déjà définis, sous forme de **hiérarchie**. Dans ce cas, la signature totale est la réunion des signatures de tous les TAA.

Si des opérateurs utilisent le même symbole, le problème de **surcharge** peut être résolu sans difficulté, parce que les opérateurs sont définis par leur ensembles de définitions.

SYNTAXE DE L'ECRITURE D'UN TYPE ABSTRAIT ALGEBRIQUE :

sorte : .....
utilise : .....
opérations :
préconditions :
..... **def_ssi** .....
axiomes :

FinTAA

Exemple d'écriture d'un TAA (les booléens) :

sorte : Booléens
opérations :
V : → Booléens
F : → Booléens
¬ : Booléens → Booléens
∧ : Booléens x Booléens → Booléens
∨ : Booléens x Booléens → Booléens
axiomes :

¬ (**V**) = **F** ; ¬ (**F**) = **V**
a ∧ **V** = **a** ; **a** ∧ **F** = **F**
a ∨ **V** = **V** ; **a** ∨ **F** = **a**

FinBooléens

1.3 Le Type Abstrait de Donnée (TAD)

Dans la suite du document les TAA ne seront pas utilisés entièrement, la partie axiomes étant occultée. Seules les parties opérations et préconditions sont étudiées en vue de leur implantation.
C'est cette restriction d'un TAA que nous appellerons un type abstrait de données (TAD).
Nous allons fournir dans les paragraphes suivants quelques Types Abstrait de Données différents.

Nous écrirons ainsi par la suite un TAD selon la syntaxe suivante :

TAD Truc

utilise : .....
Champs : .....
opérations : .....
préconditions : .....

FinTAD Truc

Le TAD Booléens s'écrit à partir du TAA Booléens :

TAD Booléens
opérations :
V : → Booléens
F : → Booléens
¬ : Booléens → Booléens
∧ : Booléens x Booléens → Booléens
∨ : Booléens x Booléens → Booléens
FinTAD Booléen

Nous remarquons que cet outil permet de spécifier des structures de données d'une manière générale sans avoir la nécessité d'en connaître l'implantation, ce qui est une caractéristique de la notion d'abstraction.

1.4 Classification hiérarchique

Nous situons, dans notre approche pédagogique de la notion d'abstraction, les TAD au sommet de la hiérarchie informationnelle :

HIERARCHIE INFORMATIONNELLE

- **1°** TYPES ABSTRAITS (les TAA,...)
- **2°** CLASSES / OBJETS
- **3°** MODULES
- **4°** FAMILLES de PROCEDURES et FONCTIONS
- **5°** ROUTINES (procédures ou fonctions)
- **6°** INSTRUCTIONS STRUCTUREES ou COMPOSEES
- **7°** INSTRUCTIONS SIMPLES (langage évolué)
- **8°** MACRO-ASSEMBLEUR
- **9°** ASSEMBLEUR (langage symbolique)
- **10°** INSTRUCTIONS MACHINE (binaire)

Nous allons étudier dans la suite 3 exemples complets de TAD classiques : la **liste linéaire**, la pile **LIFO**1, la file **FIFO**2. Pour chacun de ces exemples, il sera fourni une spécification opérationnelle en pascal, puis plus loin en Delphi.

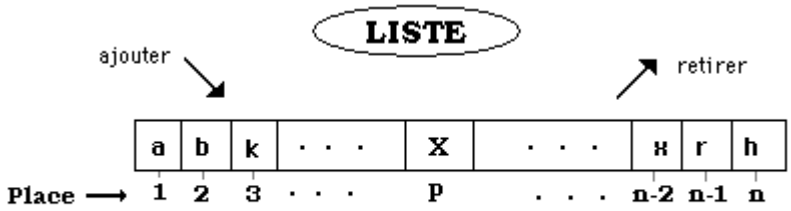
Exemples de types abstraits de données

1.5 Le TAD liste linéaire (spécifications abstraite et concrète)

Spécification abstraite

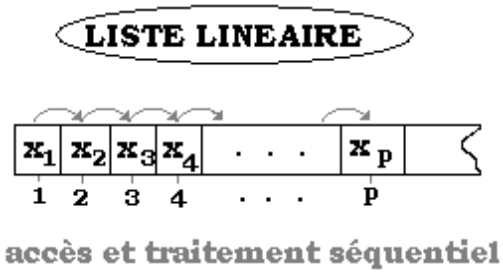
Répertorions les fonctionnalités d'une liste en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- Il est possible dans une telle structure **d'ajouter** ou de **retirer** des éléments en n'importe quel point de la liste.
- **L'ordre** des éléments est primordial. Cet ordre est construit, non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 : $(a_i)_{i \in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- Chaque place a un **contenu** de type T_0.
- Le nombre d'éléments d'une liste λ est appelé **longueur** de la liste. Si la liste est vide nous dirons que sa longueur est nulle (longueur = 0).
- On doit pouvoir effectuer au minimum (non exhaustif) les actions suivantes sur les éléments d'une liste λ : **accéder** à un élément de place fixée, **supprimer** un élément de place fixée, **insérer** un nouvel élément à une place fixée, etc ....



si Place = p **alors** Contenu (Place) = X **fsi**

- C'est une structure de donnée séquentielle dans laquelle les données peuvent être traitées les unes à la suite des autres :



De cette description nous extrayons une spécification sous forme de TAD.

Ecriture syntaxique du TAD liste linéaire

TAD Liste
utilise : **N**, T_0, Place
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
liste_vide : $\rightarrow$ Liste
acces : Liste x N $\rightarrow$ Place
contenu : Place $\rightarrow T_0$
kème : Liste x **N** $\rightarrow T_0$
long : Liste $\rightarrow$ **N**
supprimer : Liste x **N** $\rightarrow$ Liste
inserer : Liste x **N** x $\rightarrow$ Liste
succ : Place $\rightarrow$ Place

préconditions :
acces(L,k) **def_ssi** $1 \leq k \leq$ long(L)
supprimer(L,k) **def_ssi** $1 \leq k \leq$ long(L)
inserer(L,k,e) **def_ssi** $1 \leq k \leq$ long(L) +1
kème(L,k) **def_ssi** $1 \leq k \leq$ long(L)

Fin-Liste

signification des opérations : (spécification abstraite)

acces(L,k) : opération générale d'accès à la position d'un élément de rang k de la liste L.
supprimer(L,k) : suppression de l'élément de rang k de la liste L.
inserer(L,k,e) : insérer l'élément e de T_0 , à la place de l'élément de rang k dans la liste L.
kième(L,k) : fournit l'élément de rang k de la liste.

spécification opérationnelle concrète

- La liste est représentée en mémoire par un ***tableau*** et un attribut de ***longueur***.
- Le kème élément de la liste est le kème élément du tableau.
- Le tableau est plus grand que la liste (il y a donc dans cette interprétation une contrainte sur la longueur. Notons Longmax cette valeur maximale de longueur de liste).

Il faut donc, afin de conserver la cohérence, ajouter deux préconditions au **TAD Liste** :

long(**L**) **def_ssi** long(**L**) $\leq$ Longmax
inserer(**L,k,e**) **def_ssi** $(1 \leq k \leq$ long(L) +1) $\wedge$ long(**L**) $\leq$ Longmax**)**

D'autre part la structure de tableau choisie permet un traitement itératif de l'opération kème (une autre spécification récursive de cet opérateur est possible dans une autre spécification opérationnelle de type dynamique).

kème(L,k) = contenu(acces(L,k))

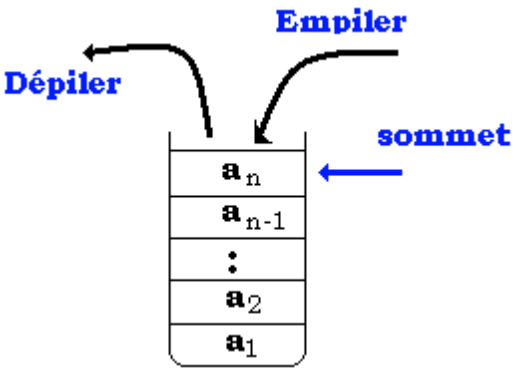
1.6 Le TAD pile LIFO (spécification abstraite et concrète)

Spécification abstraite

Répertorions les fonctionnalités d'une pile LIFO (Last In First Out) en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- C'est un modèle pour toute structure de données où l'on accumule des informations les unes après les autres, mais où l'on choisit de n'effectuer un traitement que sur le dernier élément entré. *Exemples* : pile de dossiers sur un bureau, pile d'assiettes, etc...
- Il est possible dans une telle structure **d'ajouter** ou de **retirer** des éléments uniquement au début de la pile.
- **L'ordre** des éléments est imposé par la pile. Cet ordre est construit non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste. Cet ordre n'est pas accessible à l'utilisateur, c'est un élément privé.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 :
  $(a_i)_{i \in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- La pile possède une place spéciale dénommée **sommet** qui identifie son premier élément et contient toujours le dernier élément entré.
- Le nombre d'éléments d'une pile LIFO P est appelé **profondeur** de la pile. Si la pile est vide nous dirons que sa profondeur est nulle (profondeur = 0).
- On doit pouvoir effectuer sur une pile LIFO au minimum (non exhaustif) les actions suivantes : voir si la pile **est vide**, **dépiler** un élément, **empiler** un élément, observer le **premier** élément sans le prendre, etc...

PILE LIFO



- C'est une structure de données séquentielle dans laquelle les données peuvent être traitées les unes à la suite des autres à partir du sommet

Ecriture syntaxique du TAD Pile LIFO

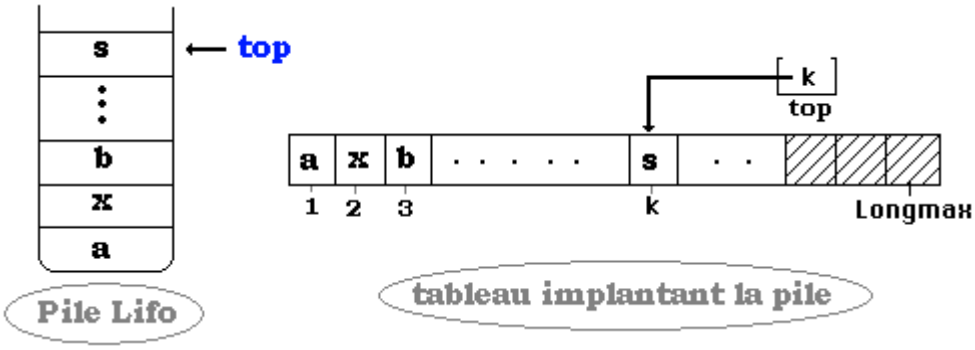
TAD PILIFO

utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
sommet : $\rightarrow$ PILIFO
Est_vide : PILIFO $\rightarrow$ Booléens
empiler : PILIFO x T_0 x sommet $\rightarrow$ PILIFO x sommet
dépiler : PILIFO x sommet $\rightarrow$ PILIFO x sommet x T_0
premier : PILIFO $\rightarrow$ T_0
préconditions :
dépiler(P) **def_ssi** est_vide(P) = **Faux**
premier(P) **def_ssi** est_vide(P) = **Faux**

FinTAD-PILIFO

spécification opérationnelle concrète

- La Pile est représentée en mémoire dans un *tableau*.
- Le *sommet* (noté **top**) de la pile est un *pointeur* sur la case du tableau contenant le début de pile. Les variations du contenu k de top se font au gré des empilements et dépilements.
- Le tableau est plus grand que la pile (il y a donc dans cette interprétation une contrainte sur la longueur, notons *Longmax* cette valeur maximale de profondeur de la pile).
- L'opérateur *empiler* : rajoute dans le tableau dans la case pointée par top un élément et top augmente d'une unité.
- L'opérateur *depiler* : renvoie l'élément pointé par **top** et diminue top d'une unité.
- L'opérateur *premier* fournit une copie du sommet pointé par top (la pile reste intacte).
- L'opérateur *Est_vide* teste si la pile est vide (vrai si elle est vide, faux sinon).

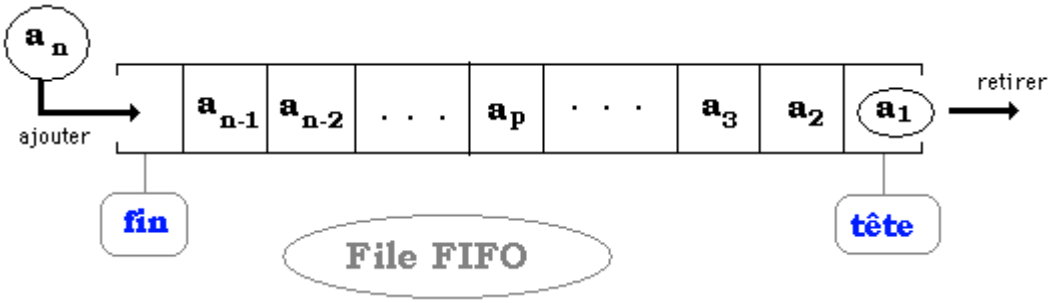


1.7 Le TAD file FIFO (spécification abstraite seule)

Spécification abstraite

Répertorions les fonctionnalités d'une file FIFO (First In First Out) en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- C'est un modèle pour toute structure de données où l'on accumule des informations les unes après les autres, mais où l'on choisit d'effectuer un traitement selon l'ordre d'arrivée des éléments, comme dans une file d'attente.
- *Exemples* :toutes les files d'attente, supermarchés, cantines , distributeurs de pièces, etc...
- Il est possible dans une telle structure **d'ajouter** des éléments à la fin de la file, ou de **retirer** des éléments uniquement au début de la file.
- **L'ordre** des éléments est imposé par la file. Cet ordre est construit non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste. Cet ordre n'est pas accessible à l'utilisateur, c'est un élément privé.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 :
  $(a_i)_{i \in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- La file possède deux places spéciales dénommées **tête** et **fin** qui identifient l'une son premier élément, l'autre le dernier élément entré.
- Le nombre d'éléments d'une file FIFO " F " est appelé **longueur** de la file ; si la file est vide nous dirons que sa longueur est nulle (longueur = 0).
- On doit pouvoir effectuer sur une file FIFO au minimum (non exhaustif) les actions suivantes : voir si la file **est vide**, **ajouter** un élément, **retirer** un élément, observer le premier élément sans le prendre, etc...



Ecriture syntaxique du TAD file FIFO

TAD FIFO
utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
  tête : $\rightarrow$ FIFO
  fin : $\rightarrow$ FIFO
  Est_vide : FIFO $\rightarrow$ Booléens
  ajouter : FIFO x T_0 x fin $\rightarrow$ PILIFO x fin
  retirer : FIFO x tête $\rightarrow$ FIFO x tête x T_0
  premier : FIFO $\rightarrow$ T_0

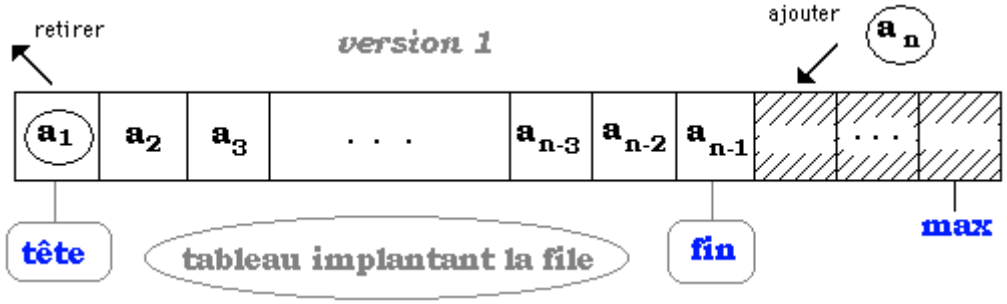
préconditions :
retirer(F) **def_ssi** est_vide(F) = **Faux**
premier(F) **def_ssi** est_vide(F) = **Faux**

FinTAD-FIFO

Spécification opérationnelle concrète

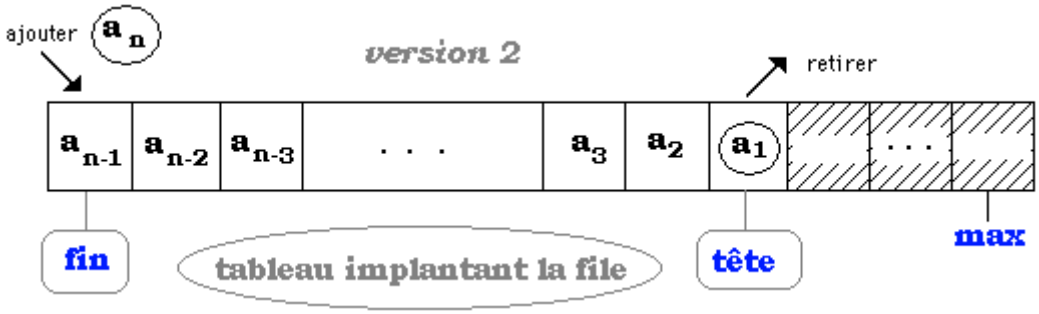
- La file est représentée en mémoire dans un *tableau*.
- La *tête* de la file est un pointeur sur la case du tableau contenant le début de la file. Les variations de la valeur de la tête se font au gré des ajouts et des retraits.
- La *fin* ne bouge pas , c'est le point d'entrée de la file.
- Le tableau est plus grand que la file (il y a donc dans cette interprétation une contrainte sur la longueur ; notons *max* cette valeur maximale de longueur de la file).
- L'opérateur *ajouter* : ajoute dans le tableau dans la case pointée par fin un élément et tête augmente d'une unité.
- L'opérateur *retirer* : renvoie l'élément pointé par tête et diminue tête d'une unité.
- L'opérateur *premier* fournit une copie de l'élément pointé par tête (la file reste intacte).
- L'opérateur *Est_vide* teste si la file est vide (vrai si elle est vide, faux sinon).

On peut ajouter après la dernière cellule pointée par l'élément fin comme le montre la figure ci-dessous :



dans ce cas retirer un élément en tête impliquera un décalage des données vers la gauche.

On peut aussi choisir d'ajouter à partir de la première cellule comme le montre la figure ci-dessous :



dans ce cas ajouter un élément en fin impliquera un décalage des données vers la droite.

4.2 : Types abstraits de données

Implantation avec Unit en Delphi

Plan du chapitre:

1. Types abstraits de données et Unités en Delphi

1.1 Traduction générale TAD → Unit pascal
1.2 Exemples de Traduction TAD → Unit pascal
1.3 Variations sur les spécifications d'implantation
1.4 Exemples d'implantation de la liste linéaire
1.5 Exemples d'implantation de la pile LIFO
1.6 Exemples d'implantation de la file FIFO

1. Types abstraits de données et Unités en Delphi

Dans cette partie nous adoptons un point de vue pratique dirigé par l'implémentation dans un langage accessible à un débutant des notions de type abstrait de donnée.

Nous allons proposer une écriture des TAD avec des unités Delphi (pascal version avec **Unit**) puis après avec des classes Delphi. Le langage Delphi en effet pour implanter des TAD, possède deux notions situées à deux niveaux d'abstraction différents :

- La notion d'unité (**Unit**) se situe au niveau 4 de la hiérarchie informationnelle citée auparavant : elle nous a déjà servi à implanter la notion de module. Une unité est donc une approximation relativement bonne pour le débutant de la notion de TAD.

- La notion classique de **classe**, contenue dans tout langage orienté objet, se situe au niveau 2 de cette hiérarchie **constitue une meilleure approche de la notion de TAD**.

En fait un TAD sera bien décrit par la partie **interface** d'une **unit** et se traduit presque immédiatement ; le travail de traduction des préconditions est à la charge du programmeur et se trouvera dans la partie *privée* de la unit (**implementation**)

1.1 Traduction générale TAD → Unit pascal

Nous proposons un tableau de correspondance pratique entre la signature d'un TAD et la partie interface d'une Unit :

syntaxe du TAD	*squelette de la unit associée*
TAD Truc	**Unit Truc ;**
utilise TAD_1 , TAD_2,...,TAD_n	**interface** **uses** TAD_1,...,TAD_n ;
champs **.....**	**const ....** **type ....** **var ....**
opérations Op1 : E x F → G Op2 : E x F x G → H x S	**procedure** Op1(x :E ;y :F ;**var** z :G) ; **procedure** Op2(x :E ;y :F ; z :G ; **var** t :H ; **var** u :S) ; **.....**
FinTAD-Truc	**end.**

Reste à la charge du programmeur l'écriture, dans la partie **implementation** de la **unit,** des blocs de code associés à chacune des procédures décrites dans la partie **interface** :

```
Unit Truc ;
interface
   uses TAD1,...,TADn ;
   const ....
   type ....
   var ....
   procedure Op1(x :E ;y :F ;var z :G) ;
   procedure Op2(x :E ;y :F ; z :G ; var t :H ; var u :S) ;
   ............

implementation
procedure Op1(x :E ;y :F ;var z :G) ;
begin
 ............ code
end ;

procedure Op2(x :E ;y :F ; z :G ; var t :H ;var u :S) ;
begin
 ............ code
end ;
etc...
end.
```

1.2 Exemples de Traduction TAD → Unit pascal

Le TAD Booléens implanté sous deux spécifications concrètes en pascal avec deux types scalaires différents.

Spécification opérationnelle concrète n°1

Les constantes du type Vrai, Faux sont représentées par deux constantes pascal dans un type intervalle sur les entiers.

```
Const   vrai=1 ; faux=0
type    Booleens=faux..vrai ;
```

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens **Champs** : **Opérations** : Vrai : → Booléens Faux : → Booléens Et : Booléens x Booléens → Booléens Ou : Booléens x Booléens → Booléens Non : Booléens → Booléens **FINTAD**-Booléens	**Unit** Bool ; **interface** **Const** vrai=1 ; faux=0 **type** Booleens = faux..vrai ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;

Spécification opérationnelle concrète n°2

Les constantes du type Vrai, Faux sont représentées par deux identificateurs pascal dans un type énuméré.

type Booleens=(faux,vrai) ;

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens **Champs** : **Opérations** : Vrai : → Booléens Faux : → Booléens Et : Booléens x Booléens → Booléens Ou : Booléens x Booléens → Booléens Non : Booléens → Booléens **FINTAD**-Booléens	**Unit** Bool ; **interface** **type** Booleens=(faux,vrai) ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;

Nous remarquons la similarité des deux spécifications concrètes :

Implantation avec des entiers	Implantation avec des énumérés
Unit Bool ; **interface** **Const** vrai=1 ; faux=0 **type** Booleens = faux..vrai ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;	**Unit** Bool ; **interface** **type** Booleens = (faux,vrai) ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;

1.3 Variations sur les spécifications d'implantation

Cet exercice ayant été proposé à un groupe d'étudiants, nous avons eu plusieurs genres d'implantation des opérations : et, ou, non. Nous exposons au lecteur ceux qui nous ont parus être les plus significatifs :

Implantation d'après spécification concrète n°1

Fonction Et	Fonction Et
function Et (x,y :Booleens) :Booleens ; **begin** Et := x * y **end** ;	**function** Et (x,y :Booleens) :Booleens ; **begin** **if** x=Faux **then** Et :=Faux **else** Et := Vrai **end** ;

Fonction Ou	Fonction Ou

function Ou (x,y :Booleens) :Booleens ;
begin
  Ou :=x+y - x*y
end ;
	function Ou (x,y :Booleens) :Booleens ;
	begin
	  if x=Vrai **then** Ou := Vrai
	  else Ou := Faux
	end ;

Fonction Non	Fonction Non

function Non(x :Booleens) :Booleens ;
begin
  Non := 1-x
end ;
	function Ou (x,y :Booleens) :Booleens ;
	begin
	  if x=Vrai **then** Ou := Vrai
	  else Ou := Faux
	end ;

Dans la colonne de gauche de chaque tableau, l'analyse des étudiants a été dirigée par le choix de la spécification concrète sur les entiers et sur un modèle semblable aux fonctions indicatrices des ensembles. Ils ont alors cherché des combinaisons simples d'opérateurs sur les entiers fournissant les valeurs adéquates.

Dans la colonne de droite de chaque tableau, l'analyse des étudiants a été dirigée dans ce cas par des considérations axiomatiques sur une algèbre de Boole. Ils se sont servis des propriétés d'absorbtion des éléments neutres de la loi " *ou* " et de la loi " *et* ". Il s'agit là d'une structure algébrique abstraite.

Influence de l'abstraction sur la réutilisation

A cette étape du travail nous avons demandé aux étudiants quel était, s'il y en avait un, le meilleur choix d'implantation quant à sa réutilisation pour l'implantation d'après la spécification concrète n°2.

Les étudiants ont compris que la version dirigée par les axiomes l'emportait sur la précédente, car sa qualité d'abstraction due à l'utilisation de l'axiomatique a permis de la réutiliser sans aucun changement dans la partie **implementation** de la unit associée à spécification concrète n°2 (en fait toute utilisation des axiomes d'algèbre de Boole produit la même efficacité).

Conclusion :

l'abstraction a permis ici une réutilisation totale et donc un gain de temps de programmation dans le cas où l'on souhaite changer quelle qu'en soit la raison, la spécification concrète.

Dans les exemples qui suivent, la notation $\cong_{\bullet}$ indique la traduction en un squelette en langage pascal.

1.4 Exemples d'implantation de la liste linéaire

spécification proposée en pseudo-Pascal :

Liste ≅	**type** Liste=**record** t : **array[**1.. Longmax**] of** T_0; long : 0.. Longmax **end**;
liste_vide ≅	**var** L : Liste (avec L.long :=0)
acces ≅	**var** k : integer; L : liste (*adresse(L.t[k]))*
contenu ≅	**var** k : integer; L : liste *(val(adresse(L.t[k])))*
kème ≅	**var** k : integer; L : liste *(kème(L,k)* ≅ L.t[k])
long ≅	**var** L : liste *(long* ≅ L.long)
succ ≅	adresse(L.t[k])+1 *c-à-dire (L.t[k+1])*
supprimer ≅	**procedure** supprimer(**var** L : Liste ; k : 1.. Longmax); **begin** ..... **end;** *{supprimer}*
inserer ≅	**procedure** inserer(**var** L : Liste ; k : 1.. Longmax; x : T_0); **begin** ..... **end;** *{inserer}*

La précondition de l'opérateur **supprimer** peut être ici implantée par le test :

if k<=long(L) **then** ....

La précondition de l'opérateur **insérer** peut être ici implantée par le test :

if (long(L) < Longmax) **and** (k<=Long(L)+1) **then** .....

Les deux objets **acces** et **contenu** ne seront pas utilisés en pratique, car le tableau les implante automatiquement d'une manière transparente pour le programmeur.

Le reste du programme est laissé au soin du lecteur qui pourra ainsi se construire sur sa machine , une base de types en Pascal-Delphi de base.

Nous pouvons " **enrichir** " le TAD Liste en lui adjoignant deux opérateurs test et rechercher (rechercher un élément dans une liste). Ces adjonctions ne posent aucun problème. Il suffit pour cela de rajouter au TAD les lignes correspondantes :

opérations
Test : Liste x $T_0 \rightarrow$ Booléen
rechercher : Liste x $T_0 \rightarrow$ Place
précondition
rechercher(**L**,**e**) **def_ssi** Test(**L**,**e**) = **V**

Le lecteur construira à titre d'exercice l'implantation Pascal-Delphi de ces deux nouveaux opérateurs en étendant le programme déjà construit. Il pourra par exemple se baser sur le schéma de représentation Pascal suivant :

```
function Test(L : liste; e : T0):Boolean;
begin
{il s'agit de tester la présence ou non de e dans la liste L}
end;

procedure rechercher(L : liste ; x : T0; var rang : integer);
begin
  if Test(L,x) then
{il s'agit de fournir le rang de x dans la liste L}
  else
rang:=0
end;
```

1.5 Exemples d'implantation de la pile LIFO

Nous allons utiliser un **tableau** avec une case supplémentaire permettant d'indiquer que le fond de pile est atteint (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en pseudo-Pascal :

Pilifo ≅	**type** Pilifo=**record** t : **array[** 0.. Longmax **]** **of** T_0; sommet: 0.. Longmax **end**;
depiler ≅	**procedure** depiler (**var** Elt : T_0 ;**var** P : Pilifo) ;
empiler ≅	**procedure** empiler(Elt : T_0 ;**var** P : Pilifo) ;

premier ≅	**procedure** premier(**var** Elt : T_0 ; P : Pilifo) ; *(on pourra utiliser une function renvoyant un T_0, si le type T_0 s'y prête !)*
Est_vide ≅	**function** Est_vide(P : Pilifo) : boolean ;

Le contenu des procédures et des fonctions est laissé au lecteur à titre d'exercice.

Remarque :

Il est aussi possible de construire une spécification opérationnelle à l'aide du TAD Liste en remplaçant dans l'étude précédente le mot " tableau " par le mot " liste ". Il est vivement conseillé au lecteur d'écrire cet exercice en Delphi pour bien se convaincre de la différence entre les niveaux d'abstractions.

1.6 Exemples d'implantation de la file FIFO

Nous allons utiliser ici aussi un **tableau** avec une case supplémentaire permettant d'indiquer que la file est vide (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en pseudo-Pascal :

Fifo ≅	**type** Fifo=**record** t : **array[** 0.. Longmax **] of** T_0; sommet: 0.. Longmax **end**;
retirer ≅	**procedure** retirer(**var** Elt : T_0 ;**var** F : Fifo) ;
ajouter ≅	**procedure** ajouter(Elt : T_0 ;**var** F : Fifo) ;
premier ≅	**procedure** premier(**var** Elt : T_0 ; P : Fifo) ; *(on pourra utiliser une function renvoyant un T_0, si le type T_0 s'y prête !)*
Est_vide ≅	**function** Est_vide(P : Fifo) : boolean ;

Le contenu des procédures et des fonctions est laissé au lecteur à titre d'exercice.

Remarque :

Comme pour le TAD Pilifo, il est aussi possible de construire une spécification opérationnelle du TAD FIFO à l'aide du TAD Liste en remplaçant dans l'étude précédente le mot " tableau " par le mot " liste ".

Une solution d'implantation de la liste linéaire en Unit Delphi

```
Unit UListchn;

 interface
const
  max_elt = 100;
type
  T0 = integer;
  liste =
  record
    suite: array[1..max_elt] of T0;
    long: 0..max_elt;
    init_ok:char;
  end;

  function longueur (L: liste): integer;
  procedure supprimer (var L: liste; k: integer);
  procedure inserer (var L: liste; k: integer; x: T0);
  function kieme (L: liste; n: integer): T0;
  function Test (L: liste; x: T0): boolean;
  procedure Rechercher (L: liste; x: T0; var place: integer);

implementation

procedure init_liste(var L:liste);
  {initialisation obligatoire}
  begin
    with L do
    begin
      long:=0;
      init_ok:='#'
    end
  end;

  function Est_vide(L:liste):boolean;
  begin
    if L.init_ok<>'#' then
    begin
      writeln('>>> Gestionnaire de Liste: Liste non
initialisée !! (erreur fatale)');
      halt
    end
    else
      if L.long=0 then
      begin
        Est_vide:=true;
        writeln('>>> Gestionnaire de Liste: Liste vide')
      end
      else
        est_vide:=false
  end;

function longueur (L: liste): integer;
  begin
    longueur := L.long
  end;

  procedure supprimer (var L: liste; k: integer);
  var
    n: 0..max_elt;
    i: 1..max_elt;
    begin
      if not Est_vide(L) then
        if k>1 then
        begin
          n := longueur(L);
          if (1<=k)and(k <= n) then
          begin
            for i := k to n - 1 do
              L.suite[i] := L.suite[i + 1];
            L.long := n - 1
          end
        end
        else
          l.long :=0;
    end;{supprimer}
```

```
procedure inserer (var L: liste; k: integer; x: T0);
 var
  n: 0..max_elt;
  i: 1..max_elt;
  begin
   if not Est_vide(L) then
   begin
    n := longueur(L);
    if (n < max_elt) and (k <= n + 1) then
    begin
     for i := n downto k do
      L.suite[i + 1] := L.suite[i];
     L.suite[k] := x
    end;
    L.long := L.long + 1
   end
   else
   begin
    L.suite[1]:=x;
    L.long :=1
   end
  end;

 function kieme (L: liste; n: integer): T0;
 begin
  if not Est_vide(L) then
  begin
   kieme := L.suite[n]
  end
 end;
```

```
procedure Test_Recherche(L:liste;x:T0;var
trouve:boolean;var rang:integer);
 var
  fini,present:boolean;
  i,n:integer;
  begin
   if not Est_vide(L) then
   begin
    fini := false;
    i := 1;
    n:=L.long;
    present := false;
    while not fini and not present do
    begin
     if i <= n then
      if L.suite[i] <> x then
       i := i + 1
      else
       present := true
     else
      fini := true
    end;
    if present then
    begin
     {valeur x trouvée a l'indice:i}
     trouve:=true;
     rang:=i
    end
    else
    begin
     {cette valeur x n'est pas dans le tableau}
     trouve:=false;
     {on n'affecte aucune valeur a rang !! }
    end
   end
  end;
```

```
function Test (L: liste; x: T0): boolean;
 {teste la presence ou non de x dans la liste L}
 var
  present:boolean;
  rang:integer;
  begin
   if not Est_vide(L) then
   begin
    Test_Recherche(L,x,present,rang);
    Test:=present
   end
  end;
```

```
procedure Rechercher (L: liste; x: T0; var place:
integer);
 var
  present:boolean;
  begin
   if not Est_vide(L) then
   begin
    Test_Recherche(L,x,present,place);
    if not present then
     place:=0
   end
  end;
end. { fin de la Unit UListchn }
```

Une solution d'implantation de la pile Lifo en Unit Delphi

```
Unit Upilifo;

 interface

const
  max_elt = 100;
  fond = 0;
type
  T0 = integer;
  pile =
  record
    suite: array[0..max_elt] of T0;
    sommet: 0..max_elt;
  end;

  function Est_Vide(P:pile):boolean;
  procedure Empiler(elt:T0;var P:pile);
  procedure Depiler(var elt:T0;var P:pile);
  function premier (P: pile): T0;

implementation

  function Est_Vide(P:pile):boolean;
  begin
    if P.sommet = fond then
      Est_Vide := true
    else
      Est_Vide := false
  end;

  procedure Empiler(elt:T0;var P:pile);
  begin
    P.sommet := P.sommet + 1;
    P.suite[P.sommet] := elt
  end;

  procedure Depiler(var elt:T0;var P:pile);
  begin
    if not Est_Vide(P) then
    begin
      elt := P.suite[P.sommet];
      P.sommet := P.sommet - 1
    end
    {la precondition est implantee ici par
     le test if not est_vide(p) then ...    }
  end;

  function premier (P: pile): T0;
  begin
    if not Est_Vide(P) then
      premier := P.suite[P.sommet]
    { la precondition est implantee ici par
      le test if not est_vide(p) then ...    }
  end;

end. { fin de la Unit Upilifo }
```

4.3 : Structures d'arbres binaires

Plan du chapitre:

1. Notions générales sur les arbres

1.1 Vocabulaire *employé sur les arbres :*

- Etiquette Racine, noeud, branche, feuille
- Hauteur, profondeur ou niveau d'un noeud
- Chemin d'un noeud , Noeuds frères, parents, enfants, ancêtres
- Degré d'un noeud
- Hauteur ou profondeur d'un arbre
- Degré d'un arbre
- Taille d'un arbre

1.2 Exemples et implémentation d'arbre

- Arbre de dérivation
- Arbre abstrait
- Arbre lexicographique
- Arbre d'héritage
- Arbre de recherche

2. Arbres binaires

2.1 TAD d'arbre binaire
2.2 Exemples et implémentation d'arbre

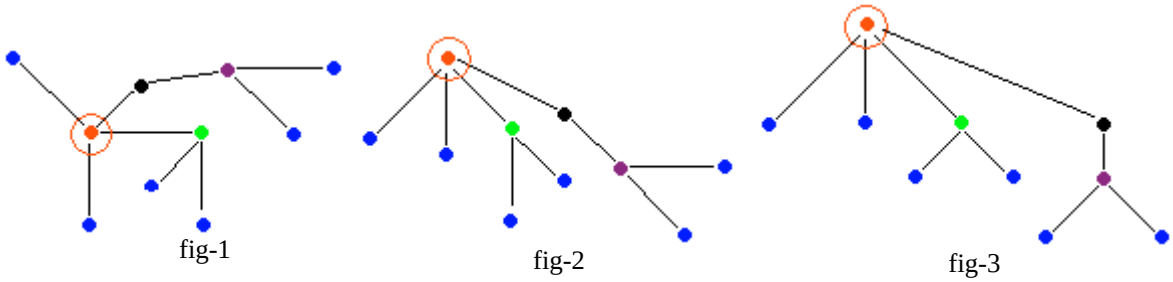
- *tableau statique*
- *variable dynamique*
- *classe*

2.3 Arbres binaires de recherche
2.4 Arbres binaires partiellement ordonnés *(tas)*
2.5 Parcours en largeur et profondeur d'un arbre binaire

- Parcours d'un arbre
- Parcours en largeur
- Parcours préfixé
- Parcours postfixé
- Parcours infixé
- Illustration d'un parcours en profondeur complet
- Exercice

1. Notions générales sur les structures d'arbres

La structure d'arbre est très utilisée en informatique. Sur le fond on peut considérer un arbre comme une généralisation d'une liste car les listes peuvent être représentées par des arbres. La complexité des algorithmes d'insertion de suppression ou de recherche est généralement plus faible que dans le cas des listes (cas particulier des arbres équilibrés). Les mathématiciens voient les arbres eux-même comme des cas particuliers de graphes non orientés connexes et acycliques, donc contenant des sommets et des arcs :



Ci dessus 3 représentations graphiques de la même structure d'arbre : dans la figure fig-1 tous les sommets ont une disposition équivalente, dans la figure fig-2 et dans la figure fig-3 le sommet "**cerclé**" se distingue des autres.

Lorsqu'un sommet est distingué par rapport aux autres, on le dénomme **racine** et la même structure d'arbre s'appelle une **arborescence**, par abus de langage dans tout le reste du document nous utliserons le vocable **arbre** pour une **arborescence**.

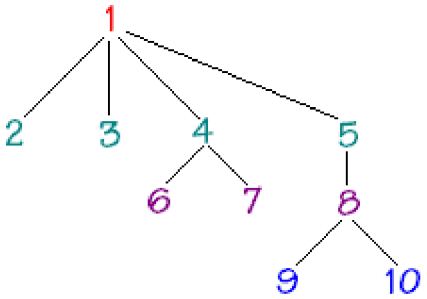
Enfin certains arbres particuliers nommés arbres binaires sont les plus utilisés en informatique et les plus simples à étudier. En outre il est toujours possible de "**binariser**" un arbre non binaire, ce qui nous permettra dans ce chapitre de n'étudier que les structures d'arbres binaires.

1.1 Vocabulaire employé sur les arbres

Etiquette

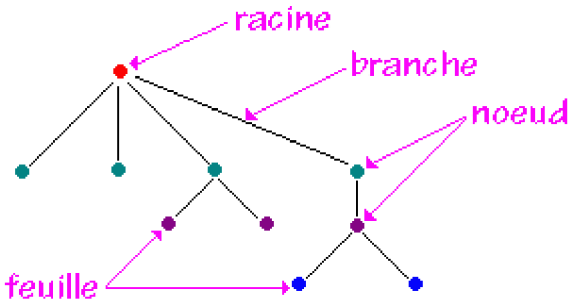
Un arbre dont tous les noeuds sont nommés est dit **étiqueté**. L'étiquette (ou nom du sommet) représente la "valeur" du noeud ou bien l'information associée au noeud.
Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :

Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :



Nous rappellons la terminologie de base sur les arbres:

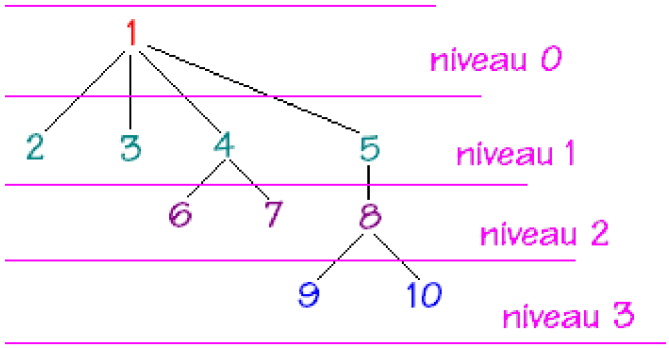
Racine , noeud, branche , feuille



Hauteur d'un noeud

Nous conviendrons de définir la **hauteur** (ou **profondeur** ou **niveau d'un noeud**) d'un noeud X comme égale **au nombre de noeuds à partir de la racine** pour aller jusqu'au noeud X.

En reprenant l'arbre précédant et en notant **h** la fonction hauteur d'un noeud :



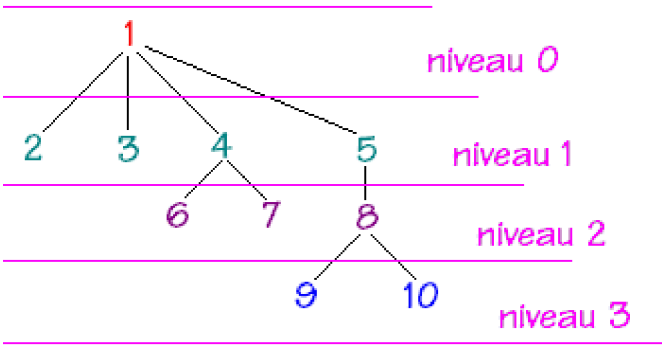
Pour atteindre le noeud étiqueté 9 , il faut parcourir le lien 1--5, puis 5--8, puis enfin 8--9 soient 4 noeuds donc 9 est de profondeur ou de hauteur égale à 4, soit **h(9) = 4**.
Pour atteindre le noeud étiqueté 7 , il faut parcourir le lien 1--4, et enfin 4--7, donc 7 est de profondeur ou de hauteur égale à 3, soit **h(7) = 3**.

Par définition la hauteur de la racine est égal à 1.
h(racine) =1 (*pour tout arbre non vide)*

(*Certains auteurs adoptent une autre convention pour calculer la hauteur d'un noeud: la racine a pour hauteur 0 et donc n'est pa comptée dans le nombre de noeuds, ce qui donne une hauteur inférieure d'une unité à notre définition*).

Chemin d'un noeud

On appelle chemin du noeud X la **suite des noeuds** par lesquels il faut passer pour aller de la racine vers le noeud X.

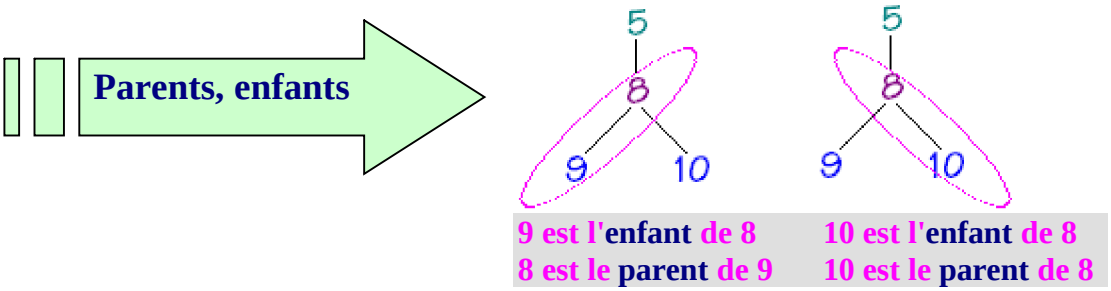


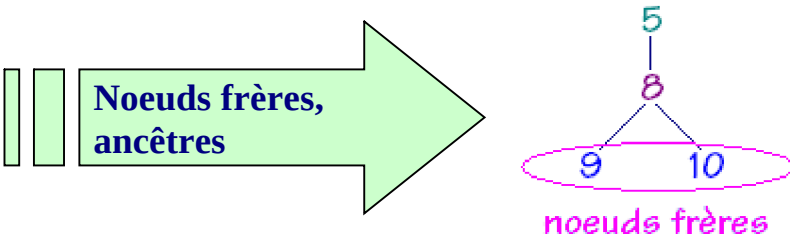
Chemin du noeud 10 = (1,5,8,10)
Chemin du noeud 9 = (1,5,8,9)
.....
Chemin du noeud 7 = (1,4,7)
Chemin du noeud 5 = (1,5)
Chemin du noeud 1 = (1)

Remarquons que la hauteur h d'un noeud X est égale au nombre de noeuds dans le chemin :

h(X) = NbrNoeud(Chemin(X)).

Le vocabulaire de lien entre noeuds de niveau différents et reliés entres eux est emprunté à la généalogie :

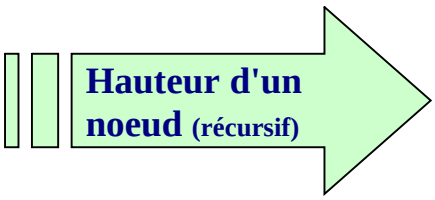




- **9 et 10 sont des frères**
- **5 est le parent de 8 et l'ancêtre de 9 et 10.**

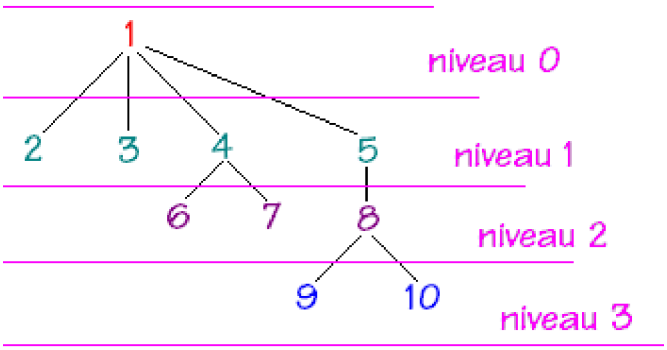
On parle aussi d'ascendant, de descendant ou de fils pour évoquer des relations entres les noeuds d'un même arbre reliés entre eux.

Nous pouvons définir récursivement la hauteur h d'un noeud X à partir de celle de son parent :

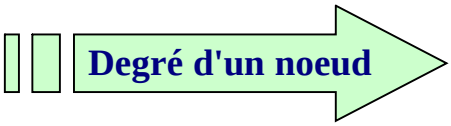


h (racine) = 1;
h (X) = 1+ h (parent (X))

Reprenons l'arbre précédent en exemple :

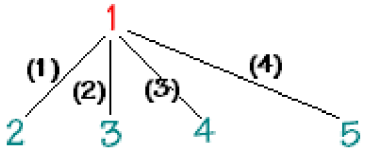


Calculons récursivement la hauteur du noeud 9, notée h(9) :
h(9) = 1+h(8)
h(8) = 1+h(5)
h(5) = 1+h(1)
h(1) = 1 = h(5)=2 = h(8)=3 = h(9)=4

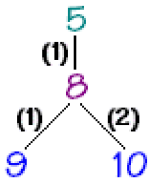


Par définition le **degré** d'un noeud est égal au **nombre de ses descendants** (enfants).

Soient les deux exemples ci-dessous extraits de l'arbre précédent :

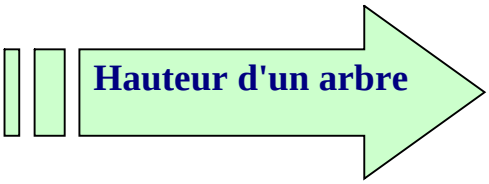


Le noeud 1 est de degré 4, car il a 4 enfants



Le noeud 5 n'ayant qu'un enfant son degré est 1.
Le noeud 8 est de degré 2 car il a 2 enfants.

Remarquons
que lorsqu'un arbre a **tous ses noeuds de degré 1**, on le nomme **arbre dégénéré** et que c'est en fait une **liste**.



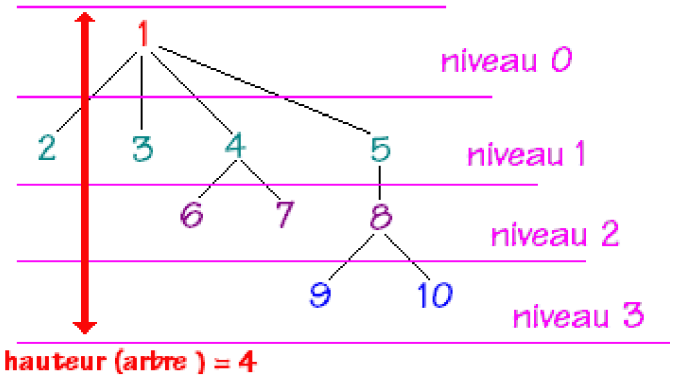
Par définition c'est le **nombre de noeuds du chemin le plus long** dans l'arbre.; on dit aussi profondeur de l'arbre.

La hauteur **h** d'un arbre correspond donc au nombre maximum de niveaux :

Hauteur d'un arbre

h (Arbre) = max { h (X) / ∀ X, X noeud de Arbre }
si Arbre = ∅ alors h(Arbre) = 0

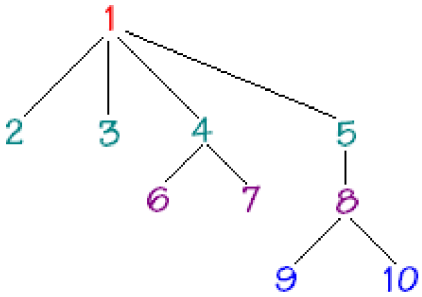
La hauteur de l'arbre ci-dessous :



Degré d'un arbre

Le degré d'un arbre est égal au **plus grand des degrés de ses nœuds :**
d°(Arbre) = max { d°(X) / ∀ X, X noeud de Arbre }

Soit à répertorier dans l'arbre ci-dessous le degré de chacun des noeuds :



d°(1) = 4 d°(2) = 0
d°(3) = 0 d°(4) = 2
d°(5) = 1 d°(6) = 0
d°(7) = 0 d°(8) = 2
d°(9) = 0 d°(10) = 0

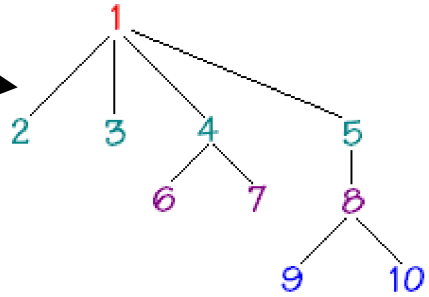
La valeur maximale est 4 , donc cet arbre est de degré 4.

Taille d'un arbre

On appelle **taille** d'un arbre le nombre total de noeuds de cet arbre :

taille(< r , fg , fd >) = 1 + taille(fg) + taille(fd)

L'arbre ci-contre a pour **taille** 10 (car il a 10 noeuds)



1.2 Exemples et implémentation d'arbre

Les structures de données arborescentes permettent de représenter de nombreux problèmes, nous proposons ci-après quelques exemples d'utilisations d'arbres dans des contextes différents.

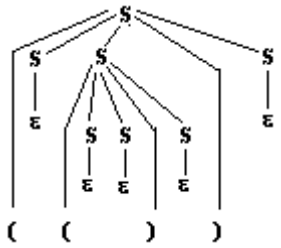
Arbre de dérivation d'un mot dans une grammaire

Exemple - 1
arbre d'analyse

Soit la grammaire G_2 : VN = {S}
V_T = {(,)}
Axiome : S
Règles
1 : S → (SS)S
2 : S → ε

Le langage $L(G_2)$ se dénomme langage des parenthèses bien formées.

Soit le mot (( )) de G_2 , voici un arbre de dérivation de (( )) dans G_2 :



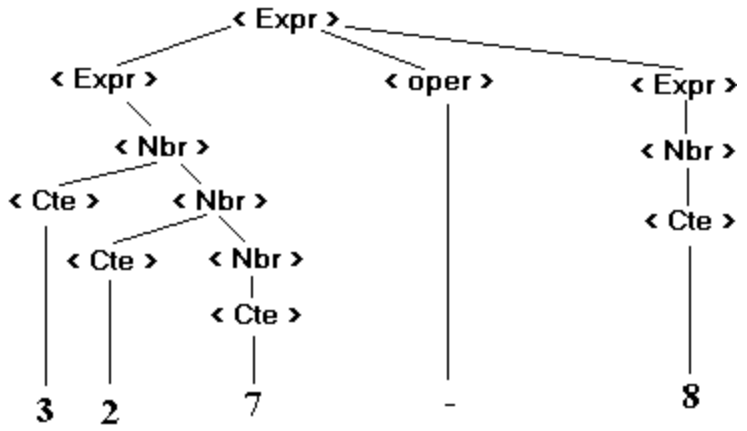
Exemple - 2
arbre abstrait

Soit la grammaire Gexp :

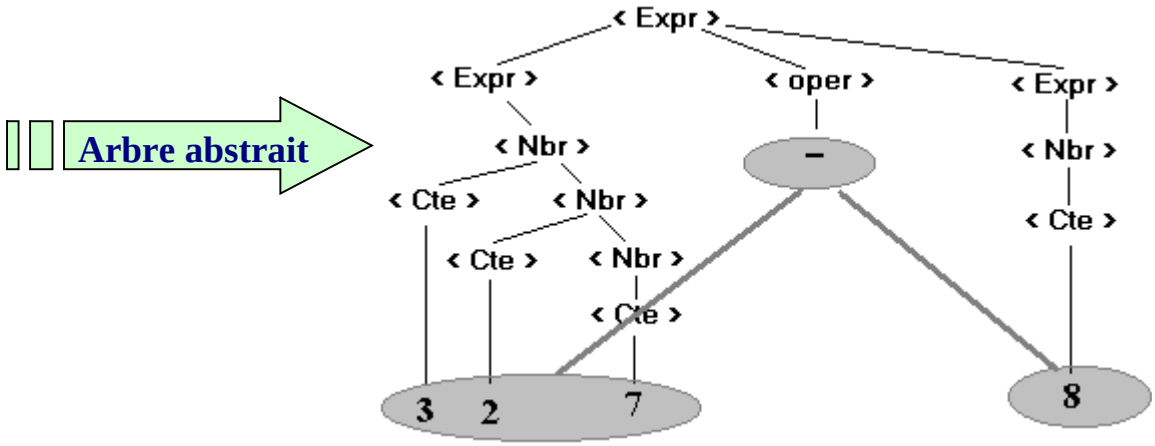
G_{exp} = (V_N,V_T,Axiome,Règles)
V_T = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(**}
V_N = { ⟨ Expr ⟩, ⟨ Nbr ⟩, ⟨ Cte ⟩, ⟨ Oper ⟩}
Axiome : ⟨ Expr ⟩
Règles :
1 :⟨ Expr ⟩→⟨ Nbr ⟩ | **(**⟨Expr ⟩**)**| ⟨ Expr ⟩⟨ Oper ⟩⟨ Expr ⟩
2 :⟨ Nbr ⟩→⟨ Cte ⟩ | ⟨ Cte ⟩⟨ Nbr ⟩
3 :⟨ Cte ⟩→**0** | **1** |...| **9**
4 :⟨ Oper ⟩→**+** | **-** | ***** | **/**

soit : **327 - 8** un mot de $L(G_{exp})$

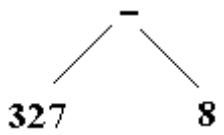
Soit son arbre de dérivation dans G_{exp} :



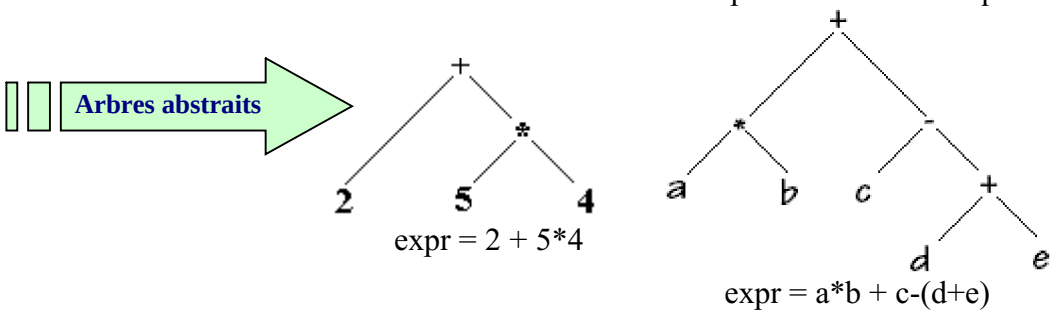
L'arbre obtenu ci-dessous en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 " :



On note ainsi cet arbre abstrait :

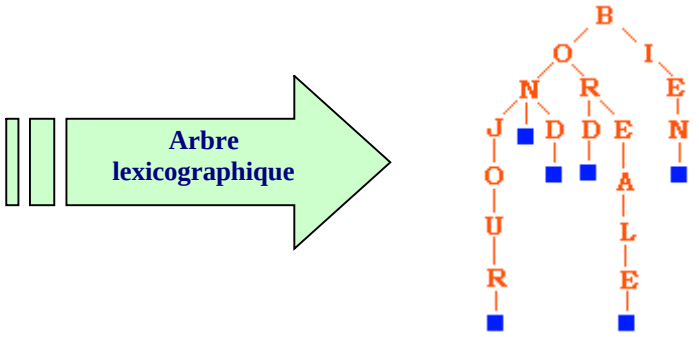


Voici d'autres abres abstraits d'expressions arithmétiques :

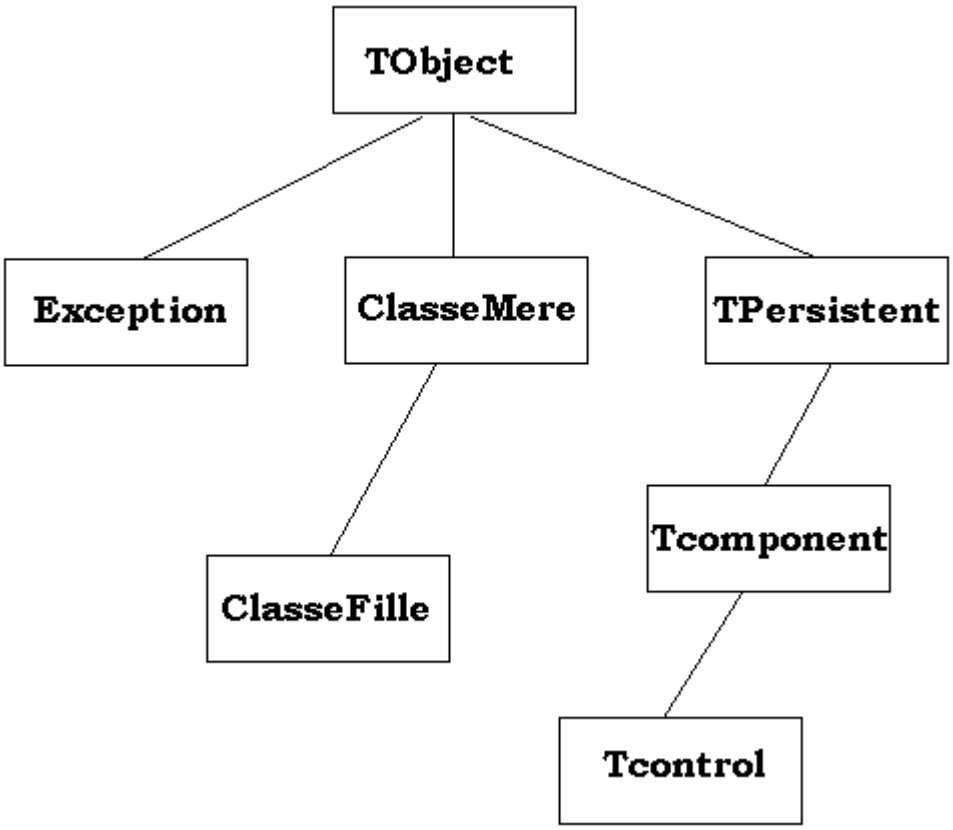


Arbre lexicographique

Rangement de mots par ordre lexical (alphabétique)
Soient les mots BON, BONJOUR, BORD, BOND, BOREALE, BIEN, il est possible de les ranger ainsi dans une structure d'arbre :



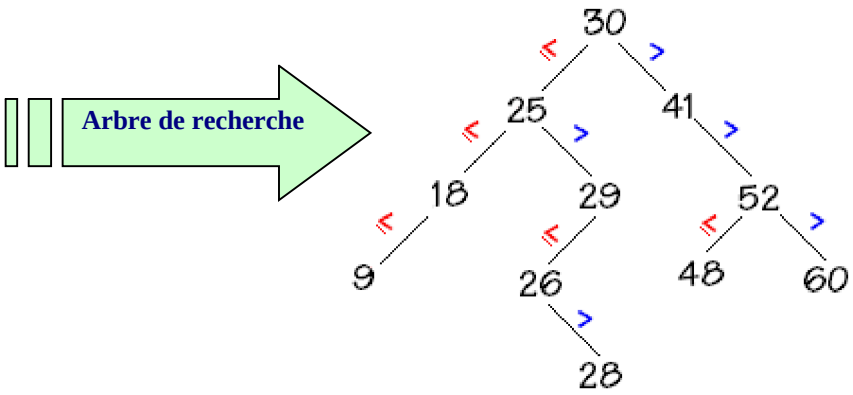
Arbre d'héritage en Delphi



Arbre de recherche

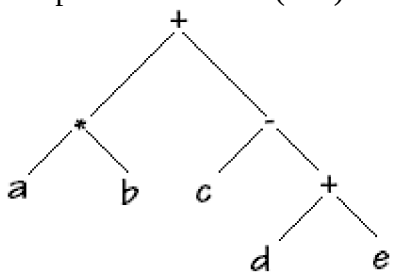
Voici à titre d'exemple que nous étudierons plus loin en détail, un arbre dont les noeuds sont de degré 2 au plus et qui est tel que pour chaque noeud la valeur de son enfant de gauche lui est inférieure ou égale, la valeur de son enfant de droite lui est strictement supérieure.

Ci-après un tel arbre ayant comme racine 30 et stockant des entiers selon cette répartition :

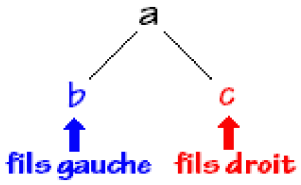


2 Les arbres binaires

Un arbre **binaire** est un arbre de degré 2 (dont les noeuds sont de degré 2 au plus).
L'arbre abstrait de l'expression a*b + c-(d+e) est un arbre binaire :



Vocabulaire :
Les descendants (enfants) d'un noeud sont lus de gauche à droite et sont appelés respectivement **fils gauche** (descendant gauche) et **fils droit** (descendant droit) de ce noeud.



Les arbres binaires sont utilisés dans de très nombreuses activités informatiques et comme nous l'avons déjà signalé il est toujours possible de reprrésenter un arbre général (de degré 2) par un arbre binaire en opérant une "binarisation".

Nous allons donc étudier dans la suite, le comportement de cette structure de donnée récursive.

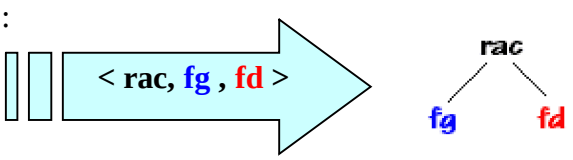
2.1 TAD d'arbre binaire

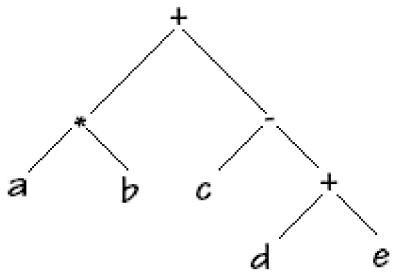
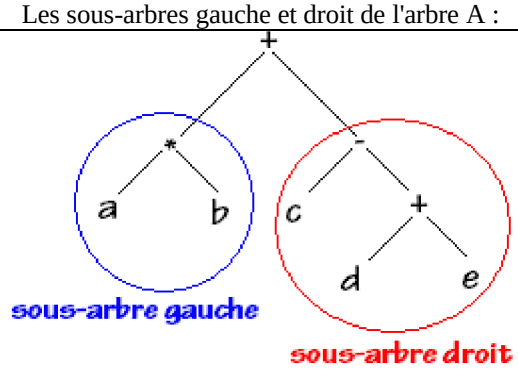
Afin d'assurer une cohérence avec les autres structures de données déjà vues (**liste, pile, file**) nous proposons de décrire une abstraction d'un arbre binaire avec un TAD. Soit la signature du TAD d'arbre binaire :

```
TAD ArbreBin
utilise : T0, Noeud, Booleens
opérations :
   ∅   : → ArbreBin
   Racine : ArbreBin → Noeud
   filsG : ArbreBin → ArbreBin
   filsD : ArbreBin → ArbreBin
   Constr : Noeud x ArbreBin x ArbreBin → ArbreBin
   Est_Vide  : ArbreBin → Booleens
   Info : Noeud → T0

préconditions :

   Racine(Arb) def_ssi Arb ≠ ∅
   filsG(Arb)  def_ssi Arb ≠ ∅
   filsD(Arb)  def_ssi Arb ≠ ∅
axiomes :

    ∀rac ∈ Noeud ,  ∀fg ∈ ArbreBin , ∀fd ∈ ArbreBin
    Racine(Constr(rac,fg,fd)) = rac
    filsG(Constr(rac,fg,fd))  = fg
    filsD(Constr(rac,fg,fd))  = fd
    Info(rac) ∈T0

FinTAD-PILIFO
```

- T_0 est le type des données rangées dans l'arbre.
- L'opérateur **filsG( ) renvoie le sous-arbre gauche** de l'arbre binaire, l'opérateur **filsD( ) renvoie le sous-arbre droit** de l'arbre binaire, l'opérateur Info( ) permet de stocker des informations de type T_0 dans chaque noeud de l'arbre binaire.

Nous noterons **< rac, fg , fd >** avec conventions implicites un arbre binaire dessiné ci-dessous :



Exemple, soit l'arbre binaire A :	Les sous-arbres gauche et droit de l'arbre A :
A = 	 filsG(A) = < * , a , b > filsD(A) = < - , c , < + , d , e > >

2.2 Exemples et implémentation d'arbre binaire étiqueté

Nous proposons de représenter un **arbre binaire étiqueté** selon deux spécifications différentes classiques :

1°) Une implantation fondée sur une structure de tableau en allocation de mémoire statique, nécessitant de connaître au préalable le nombre maximal de noeuds de l'arbre (ou encore sa taille).

2°) Une implantation fondée sur une structure d'allocation de mémoire dynamique implémentée soit par des pointeurs (variables dynamiques) soit par des références (objets) .

Implantation dans un tableau statique

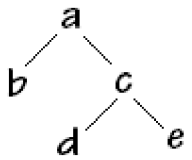
Spécification concrète
Un noeud est une structure statique contenant 3 éléments :

- l'information du noeud
- le fils gauche
- le fils droit

Pour un arbre binaire de taille = n, **chaque noeud de l'arbre binaire est stocké dans une cellule d'un tableau** de dimension 1 à n cellules. Donc chaque noeud est repéré dans le tableau par un indice (celui de la cellule le contenant).
Le champ fils gauche du noeud sera l'indice de la cellule contenant le descendant gauche, et le champ fils droit vaudra l'indice de la cellule contenant le descendant droit.

Exemple

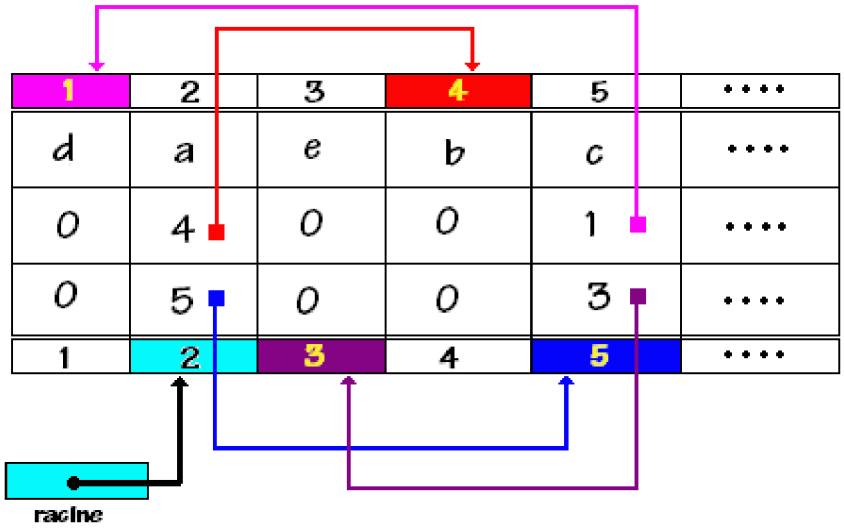
Soit l'arbre binaire ci-contre :



Selon l'implantation choisie, par hypothèse de départ, la racine <a, vers b, vers c >est contenue dans la cellule d'indice 2 du tableau, les autres noeuds sont supposés être rangés dans les cellules 1, 3,4,5 :

Nous avons :

racine = table[2]
table[1] = < d , 0 , 0 >
table[2] = < a , **4** , **5** >
table[3] = < e , 0 , 0 >
table[4] = < b , 0 , 0 >
table[5] = < c , **1** , **3** >



Explications :

table[2] = < a , **4** , **5** >signifie que le fils gauche de ce noeud est dans table[**4**] et son fils droit dans table[**5**]
table[5] = < c , **1** , **3** >signifie que le fils gauche de ce noeud est dans table[**1**] et son fils droit dans table[**3**]
table[1] = < d , 0 , 0 > signifie que ce noeud est une feuille
...etc

Spécification d'implantation en

Nous proposons d'utiliser les déclarations suivantes :

const taille = n; **type** Noeud = **record** info : T0; filsG , filsD : 0..taille ; **end**; Tableau = Array[1..taille] of Noeud ; ArbrBin = **record** ptrRac : 0..taille; table : Tableau ; **end**; **Var** Tree : ArbrBin ;	*Explications :* Lorsque **Tree.ptrRac = 0** on dit que l'arbre est vide. L'accès à la racine de l'arbre s'effectue ainsi : **Tree.table[ptrRac]** L'accès à l'info de la racine de l'arbre s'effectue ainsi : **Tree.table[ptrRac].info** L'accès au fils gauche de la racine de l'arbre s'effectue ainsi : **var** ptr:0..taille ; **ptr := Tree.table[ptrRac].filsG**; **Tree.table[ptr]** ....

L'insertion ou la suppression d'un noeud dans l'arbre ainsi représenté s'effectue directement dans une cellule du tableau. Il faudra donc posséder une structure (de liste, de pile ou de file par exemple) permettant de connaître les cellules libres ou de ranger une cellule nouvellement libérée. Une telle structure se dénomme "**espace libre**".

L'insertion se fera dans la première cellule libre, l'espace libre diminuant d'une unité.
La suppression rajoutera une nouvelle cellule dans l'espace libre qui augmentera d'une unité.

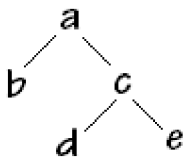
Implantation avec des variables dynamiques

Spécification concrète
Le noeud reste une structure statique contenant 3 éléments dont 2 sont dynamiques :

- l'information du noeud
- une référence vers le fils gauche
- une référence vers le fils droit

Exemple

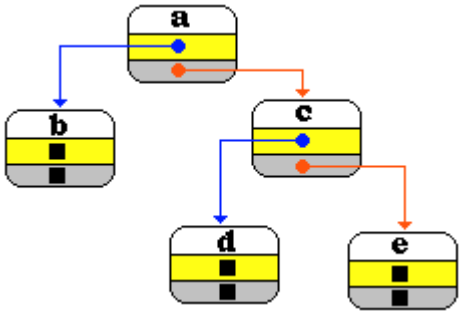
Soit l'arbre binaire ci-contre :



Selon l'implantation choisie, par hypothèse de départ, la référence vers la racine **pointe vers** la structure statique (le noeud) < a, ref vers b, ref vers c >

Nous avons :

ref racine → < a, ref vers b, ref vers c >
ref vers b → < b, null, null >
ref vers c → < a, ref vers d, ref vers e >
ref vers d → < d, null, null >
ref vers e → < e, null, null >



Spécification d'implantation en Delphi

Nous proposons d'utiliser les déclarations de variables dynamiques suivantes :

type ArbrBin = ^Noeud ; Noeud = **record** info : T0; filsG , filsD : ArbrBin ; **end**; **Var** Tree : ArbrBin ;	*Explications :* Lorsque **Tree = nil** on dit que l'arbre est vide. L'accès à la racine de l'arbre s'effectue ainsi : **Tree** L'accès à l'info de la racine de l'arbre s'effectue ainsi : **Tree^.info** L'accès au fils gauche de la racine de l'arbre s'effectue ainsi : **Tree^.filsG** L'accès au fils gauche de la racine de l'arbre s'effectue ainsi : **Tree^.filsD**

Nous noterons une simplification notable des écritures dans cette implantation par rapport à l'implantation dans un tableau statique. Ceci provient du fait que la **structure d'arbre est définie récursivement** et que la notion de variable dynamique permet une définition récursive donc plus proche de la structure.

Implantation avec une classe Delphi

Nous livrons ci-dessous une écriture de la signature et l'implémentation minimale d'une classe d'arbre binaire nommée TreeBin en Delphi (l'implémentation complète est à construire lors des exercices sur les classes) :

```
TreeBin = class
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
  end;
```

2.3 Arbres binaires de recherche

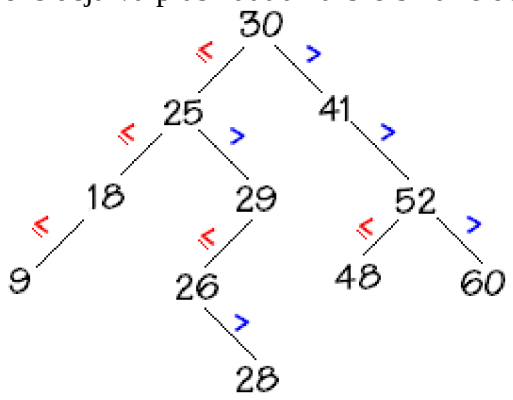
- Nous avons étudié prédédement des algorithmes de recherche en table, en particulier la recherche dichotomique dans une table triée dont la recherche s'effectue en **O(log(n))** comparaisons.
- Toutefois lorsque le nombre des éléments varie (ajout ou suppression) ces ajouts ou suppressions peuvent nécessiter des temps en **O(n)**.
- En utilisant une liste chaînée qui approche bien la structure dynamique (plus gourmande en mémoire qu'un tableau) on aura en moyenne des temps de suppression ou de recherche au pire de l'ordre de **O(n)**. L'ajout en fin de liste ou en début de liste demandant un temps constant noté **O(1)**.

Les arbres binaires de recherche sont un bon compromis pour un temps **équilibré entre ajout, suppression et recherch**e.

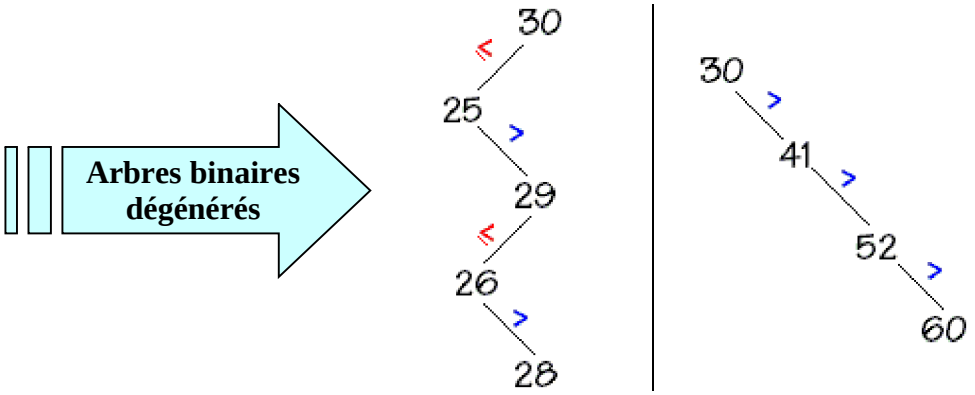
Un arbre binaire de recherche satisfait aux critères suivants :

- L'ensemble des étiquettes est **totalement ordonné**.
- Une étiquette est dénommée **clef**.
- Les **clefs** de tous les noeuds du sous-arbre **gauche** d'un noeud X, sont **inférieures ou égales** à la clef de X.
- Les **clefs** de tous les noeuds du sous-arbre **droit** d'un noeud X, sont **supérieures** à la clef de X.

Nous avons déjà vu plus haut un arbre binaire de recherche :



Prenons par exemple le noeud (25) son sous-arbre droit est bien composé de noeuds dont les clefs sont supérieures à 25 : (29,26,28). Le sous-arbre gauche du noeud (25) est bien composé de noeuds dont les clefs sont inférieures à 25 : (18,9).
On appelle arbre binaire dégénéré un arbre binaire dont le degré = 1, ci-dessous 2 arbres binaires de recherche dégénérés :

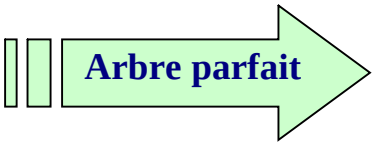


Nous remarquons dans les deux cas que nous avons affaire à une liste chaînée donc le nombre d'opération pour la suppression ou la recherche est au pire de l'ordre de O(n).

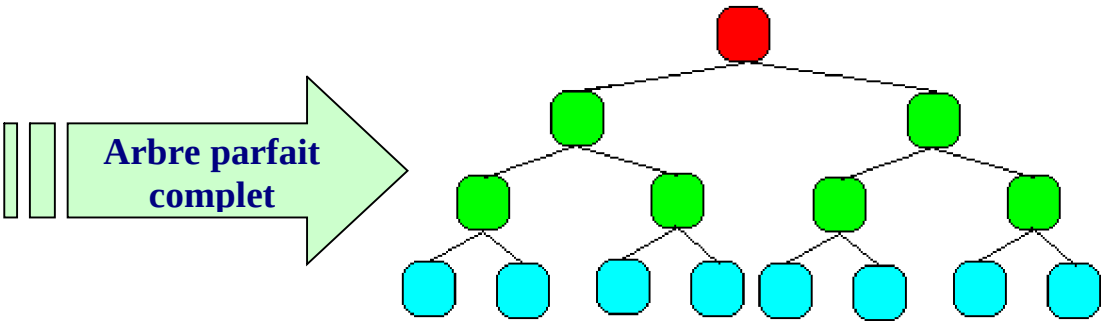
Il faudra donc utiliser une catégorie spéciale d'arbres binaires qui restent équilibrés (leurs feuilles sont sur 2 niveaux au plus) pour assurer une recherche au pire en O(log(n)).

2.4 Arbres binaires partiellement ordonnés (tas)

Nous avons déjà évoqué la notion d'arbre parfait lors de l'étude du tri par tas, nous récapitulons ici les éléments essentiels le lecteur

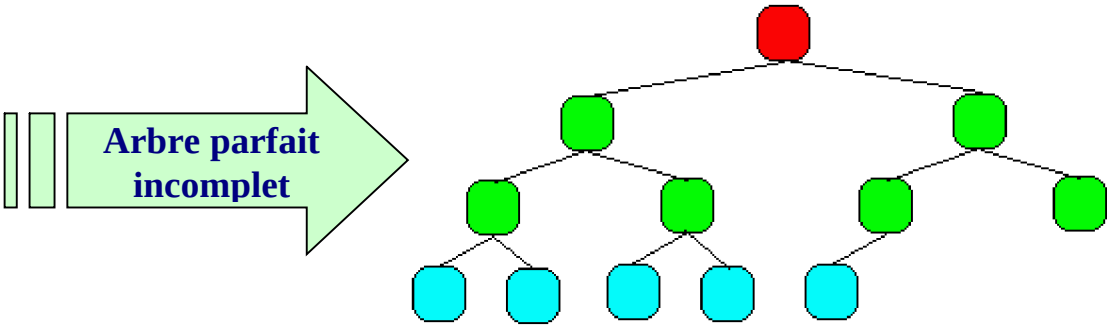


c'est un arbre binaire dont tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles), dans ce cas l'arbre parfait est un arbre binaire incomplet et les feuilles du dernier niveau **doivent être regroupées à partir de la gauche** de l'arbre.



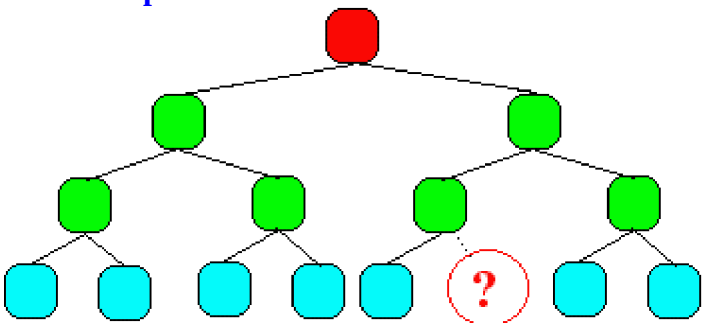
parfait complet : *le dernier niveau est complet car il contient tous les enfants*

un abre **parfait** *peut être incomplet lorsque le dernier niveau de l'arbre est incomplet (dans le cas où manquent des feuilles à la droite du dernier niveau, les feuilles sont regroupées à gauche)*



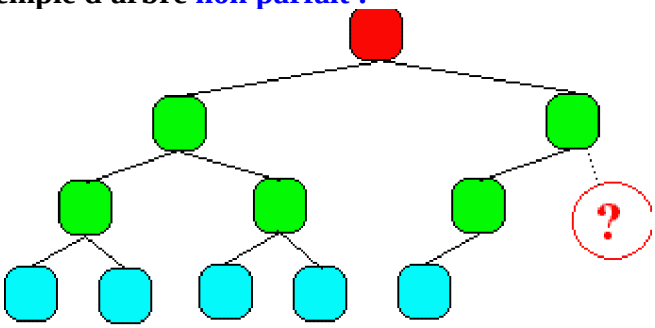
parfait icomplet: *le dernier niveau est incomplet car il manque 3 enfants à la droite*

Exemple d'arbre non parfait :



(**non parfait :** *les feuilles ne sont pas regroupées à gauche)*

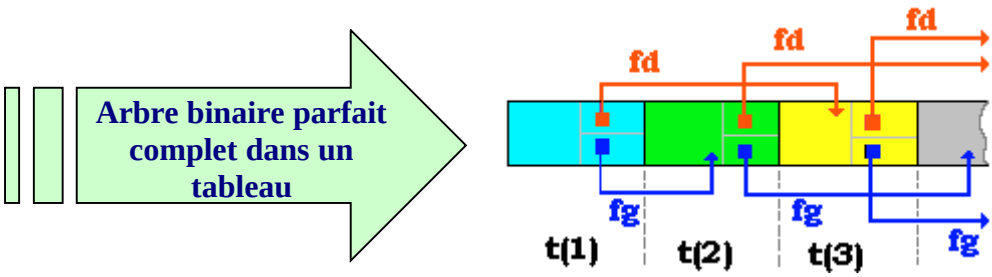
Autre exemple d'arbre non parfait :



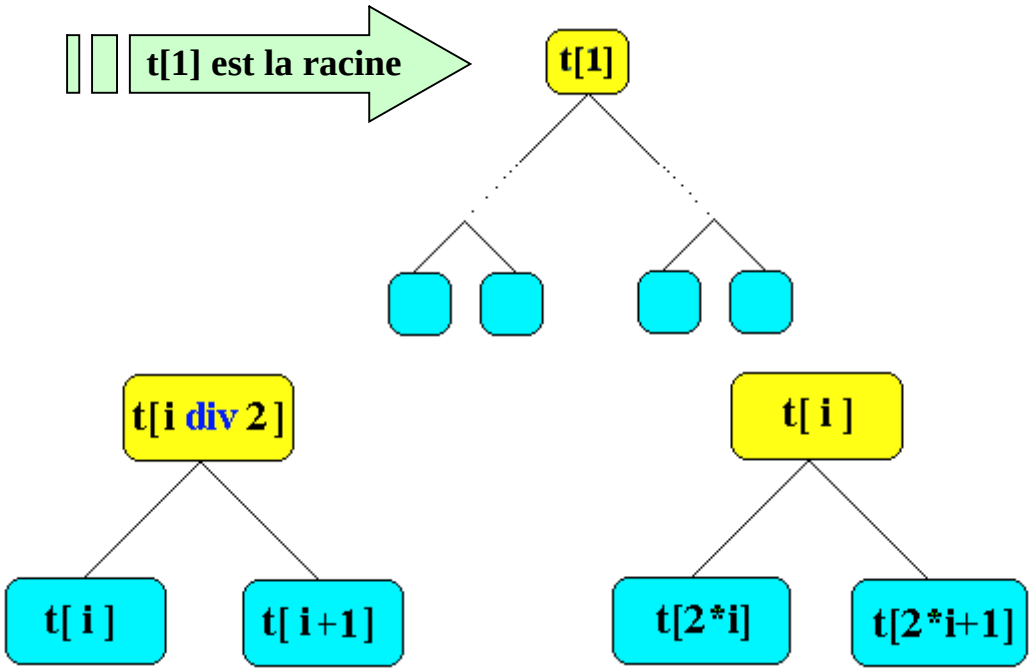
(**non parfait :** *les feuilles sont bien regroupées à gauche, mais il manque 1 enfant à l'avant dernier niveau)*

Un arbre binaire parfait se représente classiquement dans un tableau :

Les noeuds de l'arbre sont dans les cellules du tableau, il n'y a pas d'autre information dans une cellule du tableau, l'accès à la topologie arborescente est simulée à travers un calcul d'indice permettant de parcourir les cellules du tableau selon un certain 'ordre' de numérotation correspondant en fait à un **parcours hiérarchique** de l'arbre. En effet ce sont les numéros de ce parcours qui servent d'indice aux cellules du tableau nommé **t** ci-dessous :



Si **t** est ce tableau, nous avons les règles suivantes :

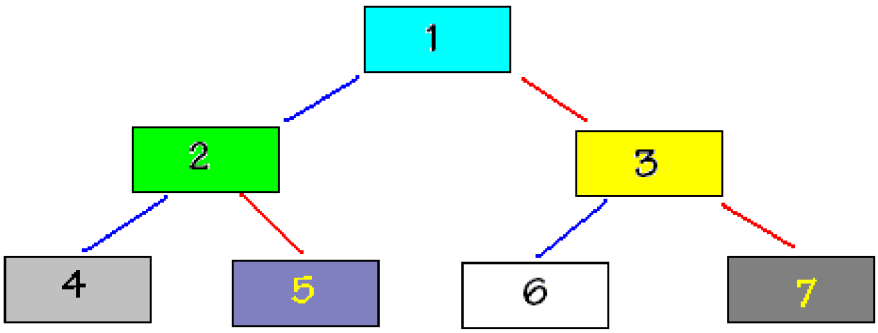


t[i **div** 2] est le père de t[i] pour i > 1

t[2 * i] et **t**[2 * i + 1] sont les deux fils, s'ils existent, de **t**[i]

si **p** est le nombre de noeuds de l'arbre et si 2 * i = p, t[i] n'a qu'un fils, t[p].
si i est supérieur à p **div** 2, t[i] est une feuille.

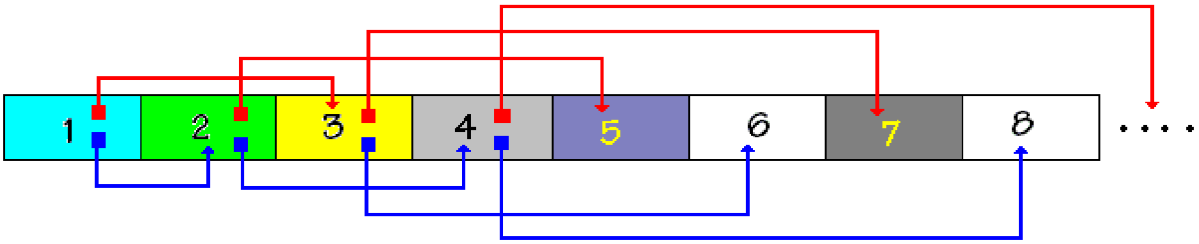
Exemple de rangement d'un tel arbre dans un tableau
(on a figuré l'indice de numérotation hiérarchique de chaque noeud dans le rectangle associé au noeud)



Cet arbre sera stocké dans un tableau en disposant séquentiellement et de façon contigüe les noeuds selon la numérotation hiérarchique (l'index de la cellule = le numéro hiérarchique du noeud).

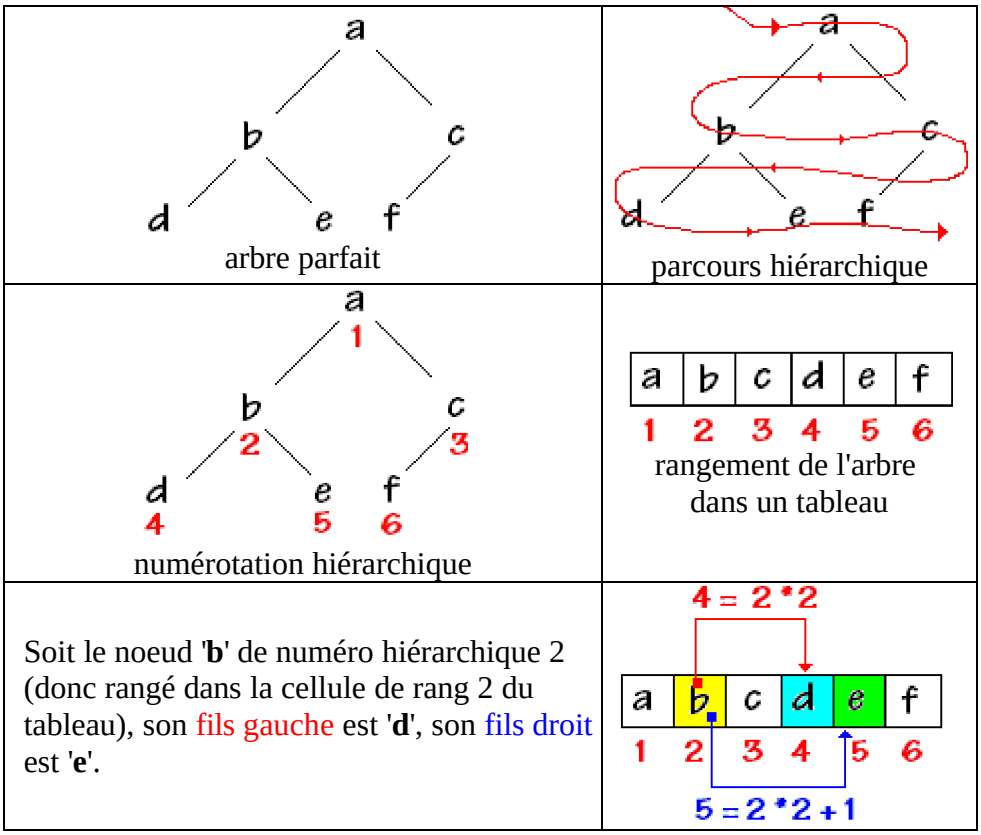
Dans cette disposition le passage d'un noeud de numéro **k** (indice dans le tableau) vers son fils gauche s'effectue par calcul d'indice, le fils gauche se trouvera dans la cellule d'index **2*k** du tableau, son fils droit se trouvant dans la cellule d'index **2*k + 1** du tableau. Ci-dessous l'arbre précédent est stocké dans un tableau : le noeud d'indice hiérarchique 1 (la racine) dans la cellule d'index 1, le noeud d'indice hiérarchique 2 dans la cellule d'index 2, etc...

Le nombre qui figure dans la cellule (nombre qui vaut l'index de la cellule = le numéro hiérarchique du noeud) n'est mis là qu'à titre pédagogique afin de bien comprendre le mécanisme.

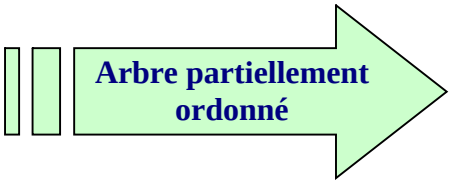


On voit par exemple, que par calcul on a bien le fils gauche du noeud d'indice 2 est dans la cellule d'index 2*2 = 4 et son fils droit se trouve dans la cellule d'index 2*2+1 = 5 ...

Exemple d'un arbre parfait étiqueté avec des caractères :



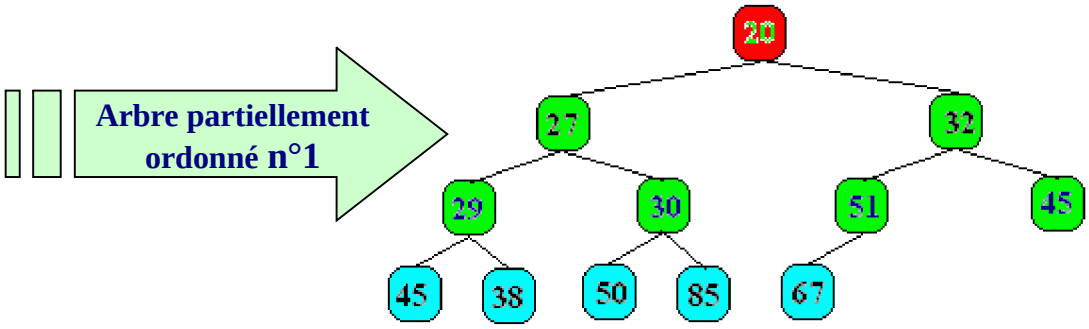
Soit le noeud '**b**' de numéro hiérarchique 2 (donc rangé dans la cellule de rang 2 du tableau), son fils gauche est '**d**', son fils droit est '**e**'.

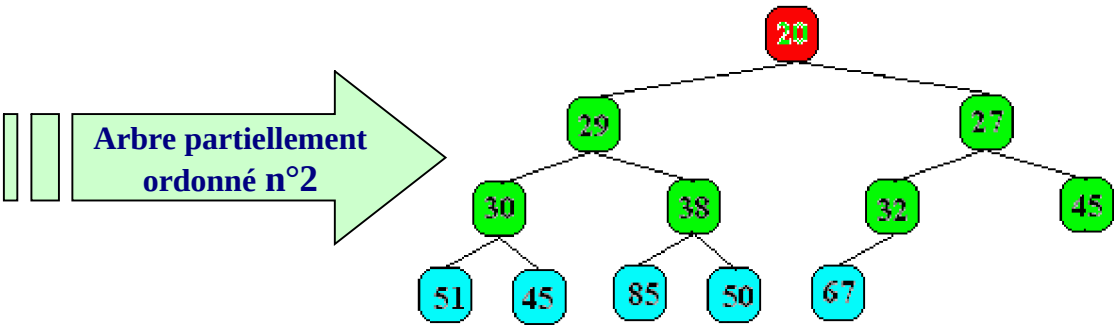


C'est un arbre étiqueté dont les valeurs des noeuds appartiennent à un ensemble muni d'une **relation d'ordre total** (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses **fils ont une valeur supérieure ou égale à celle de leur père**.

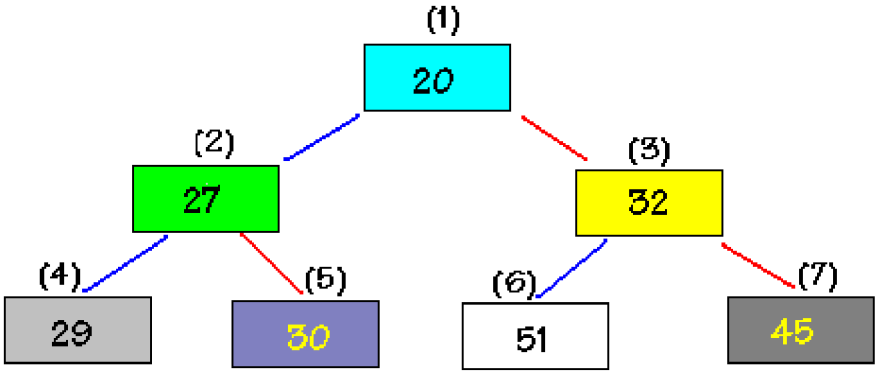
*Exemple de **deux** arbres partiellement ordonnés*

sur l'ensemble {20,27,29,30,32,38,45,45,50,51,67,85} d'entiers naturels :

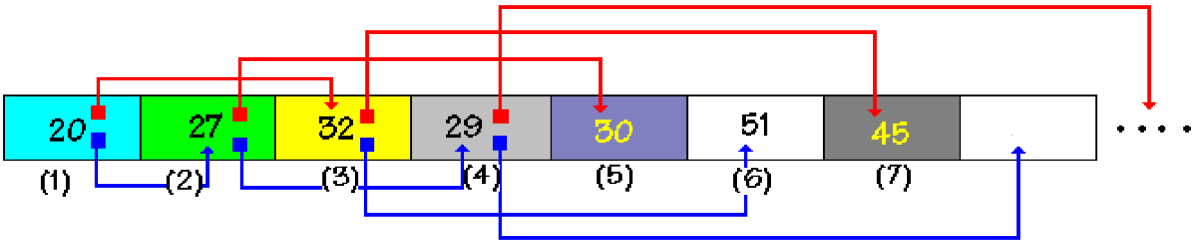




Nous remarquons que **la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum** (le plus petit élément de l'ensemble), car la valeur de ce noeud par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses descendants c'est le minimum. Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre. En reprenant l'exemple précédent sur 3 niveaux : *(entre parenthèses le numéro hiérarchique du noeud)*



Voici réellement ce qui est stocké dans le tableau : *(entre parenthèses l'index de la cellule contenant le noeud)*



Le tas

On appelle **tas** un tableau représentant un **arbre parfait partiellement ordonné**.

L'intérêt d'utiliser un arbre parfait complet ou incomplet réside dans le fait que le tableau est toujours **compacté**, les cellules vides s'il y en a se situent à la fin du tableau.
Le fait d'être partiellement ordonné sur les valeurs permet d'avoir immédiatement un **extremum** à la racine.

2.5 Parcours d'un arbre binaire

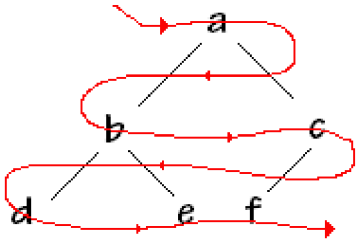
Objectif : les arbres sont des structures de données. Les informations sont contenues dans les noeuds de l'arbre, afin de construire des algorithmes effectuant des opérations sur ces informations (ajout, suppression, modification,...) il nous faut pouvoir examiner tous les noeuds d'un arbre. Examinons les différents moyens de parcourir ou de traverser chaque noeud de l'arbre et d'appliquer un traitement à la donnée rattachée à chaque noeud.

Parcours d'un arbre

L'opération qui consiste à **retrouver** systématiquement tous les noeuds d'un arbre et d'y appliquer un **même traitement** se dénomme **parcours** de l'arbre.

Parcours en largeur ou hiérarchique

Un algorithme classique consiste à **explorer** chaque noeud d'un niveau donné de **gauche à droite**, puis de passer au niveau suivant. On dénomme cette stratégie le parcours en largeur de l'arbre.



Parcours en profondeur

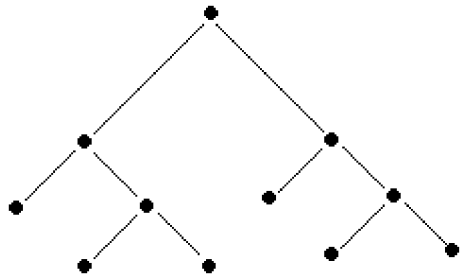
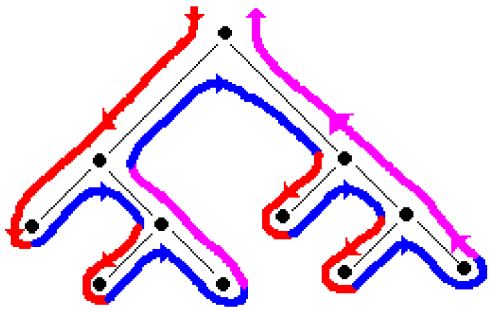
La stratégie consiste à **descendre** le plus profondément soit **jusqu'aux feuilles** d'un noeud de l'arbre, puis lorsque toutes les feuilles du noeud ont été visitées, l'algorithme "**remonte**" au noeud plus haut dont les feuilles n'ont pas encore été visitées.

Notons que ce parcours peut s'effectuer systématiquement en commençant par le fils gauche, puis en examinant le fils droit ou bien l'inverse.

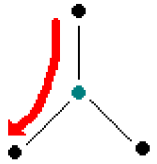
Parcours en profondeur par la gauche

Traditionnellement c'est l'exploration **fils gauche, puis ensuite fils droit** qui est retenue on dit alors que l'on traverse l'arbre en "**profondeur par la gauche**".

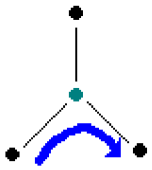
Schémas montrant le principe du parcours exhaustif en "**profondeur par la gauche**" :

Soit l'arbre binaire suivant:	Appliquons la méthode de parcours proposée :
	

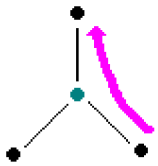
Chaque noeud a bien été examiné selon les principes du parcours en profondeur :



descendre vers le fils gauche

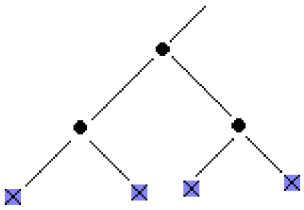


examiner le fils droit



remonter après examen des 2 fils

En fait pour ne pas surcharger les schémas arborescents, nous omettons de dessiner à la fin de chaque noeud de type feuille les deux **noeuds enfants vides** qui permettent de reconnaître que le parent est une feuille :



⊠ = arbre vide

Lorsque la compréhension nécessitera leur dessin nous conseillons au lecteur de faire figurer explicitement dans son schéma arborescent les noeuds vides au bout de chaque feuille.

Nous proposons maintenant, de donner une description en langage algorithmique LDFA du parcours en profondeur d'un arbre binaire sous forme récursive.

Algorithme général récursif de parcours en profondeur par la gauche

```
parcourir ( Arbre )
si Arbre ≠ ∅  alors
   Traiter-1 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsG ) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
   Traiter-3 (info(Arbre.Racine)) ;
Fsi
```

Les différents traitements Traiter-1 ,Traiter-2 et Traiter-3 consistent à traiter l'information située dans le noeud actuellement traversé soit lorsque l'on descend vers le fils gauche (Traiter-1), soit en allant examiner le fils droit (Traiter-2), soit lors de la remonté après examen des 2 fils (Traiter-3).

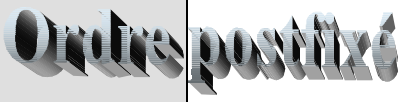
En fait on n'utilise en pratique que trois variantes de cet algorithme, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux noeuds. Chacun de ces 3 parcours définissent un ordre implicite (préfixé, infixé, postfixé) sur l'affichage et le traitement des données contenues dans l'arbre.

Algorithme de parcours en pré-ordre :

```
parcourir ( Arbre )
si Arbre ≠ ∅  alors
   Traiter-1 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsG ) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```

Algorithme de parcours en post-ordre :

```
parcourir ( Arbre )
si Arbre ≠ ∅  alors
   parcourir ( Arbre.filsG ) ;
   parcourir ( Arbre.filsD ) ;
   Traiter-3 (info(Arbre.Racine)) ;
Fsi
```



Algorithme de parcours en ordre symétrique :

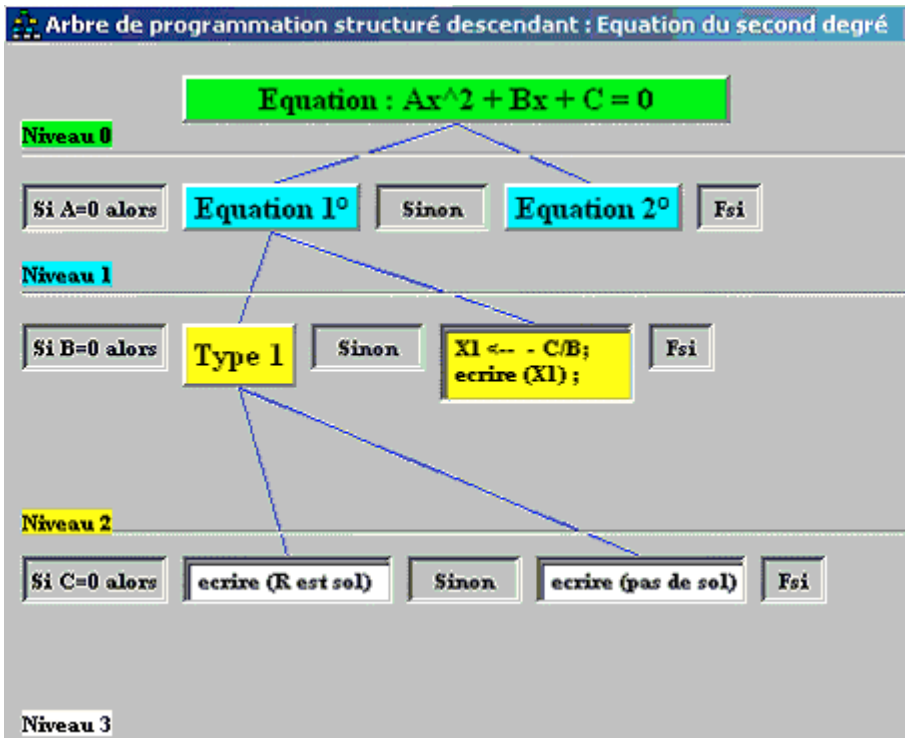
```
parcourir ( Arbre )
si Arbre  ≠ ∅  alors
   parcourir ( Arbre.filsG) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```

Illustration prtaique d'un parcours général en profondeur

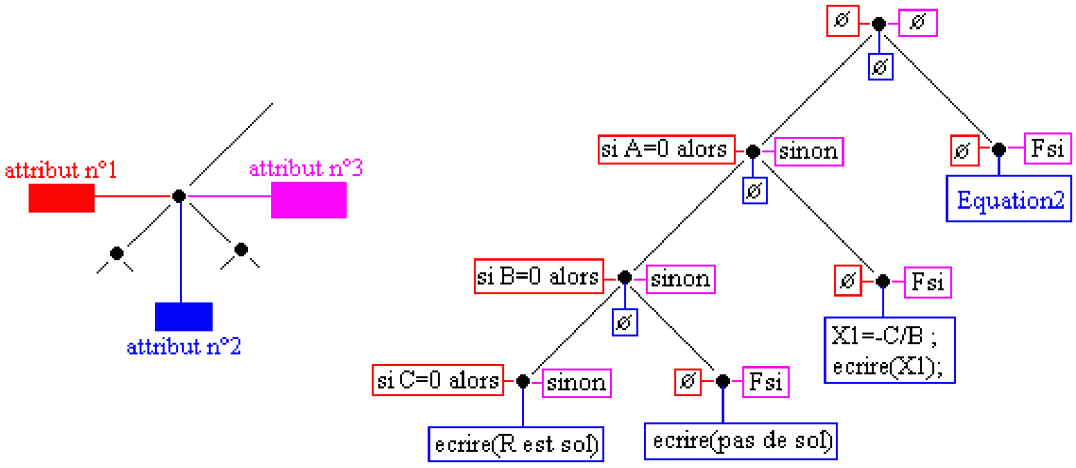
Le lecteur trouvera plus loin des exemples de parcours selon l'un des 3 ordres infixé, préfixé, postfixé, nous proposons ici un exemple didactique de parcours général avec les 3 traitements.

Nous allons voir comment utiliser une telle structure arborescente afin de restituer du texte algorithmique linéaire en effectuant un parcours en profondeur.

Voici ce que nous donne une analyse descendante du problème de résolution de l'équation du second degré (nous avons fait figurer uniquement la branche gauche de l'arbre de programmation) :



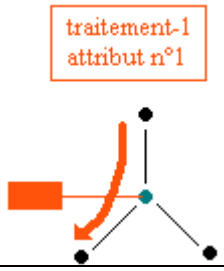
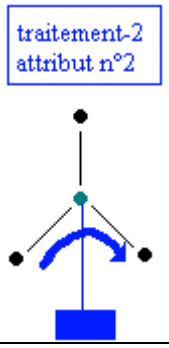
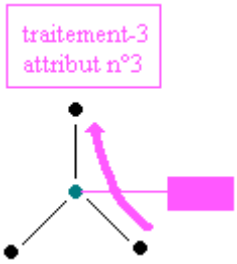
Ci-dessous une représentation schématique de la branche gauche de l'arbre de programmation précédent :



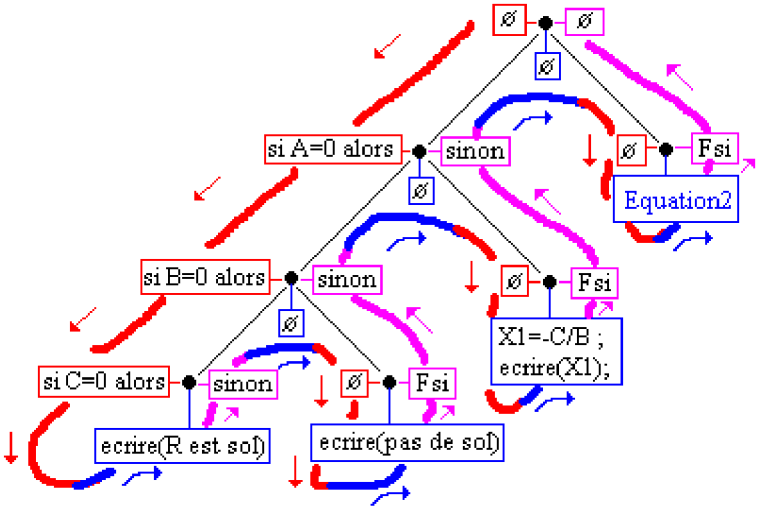
Nous avons établi un modèle d'arbre (binaire ici) où les informations au noeud sont au nombre de 3 (nous les nommerons attribut n°1, attribut n°2 et attribut n°3). Chaque attribut est une **chaîne de caractères**, vide s'il y a lieu.

Nous noterons ainsi un attribut contenant une chaîne vide : ∅

Traitement des attributs pour produire le texte

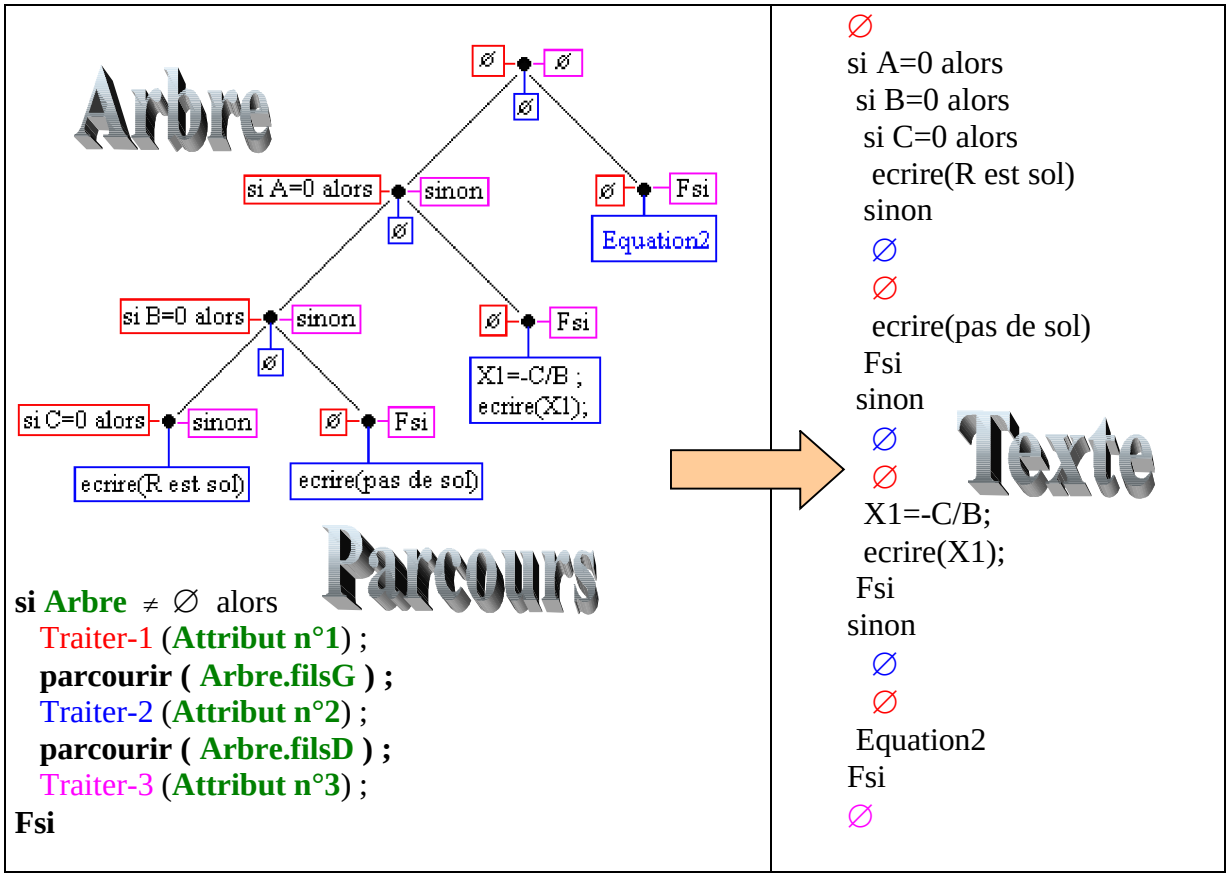
	Traiter-1 (**Arbre.Racine.Attribut n°1**) consiste à écrire le contenu de l'Attribut n°1 : **si** Attribut n°1 non vide **alors** ecrire(Attribut n°1) **Fsi**
	Traiter-2 (**Arbre.Racine.Attribut n°2**) consiste à écrire le contenu de l'Attribut n°2 : **si** Attribut n°2 non vide **alors** ecrire(Attribut n°2) **Fsi**
	Traiter-3 (**Arbre.Racine.Attribut n°3**) consiste à écrire le contenu de l'Attribut n°3 : **si** Attribut n°3 non vide **alors** ecrire(Attribut n°3) **Fsi**

Parcours en profondeur de l'arbre de programmation de l'équation du second degré :



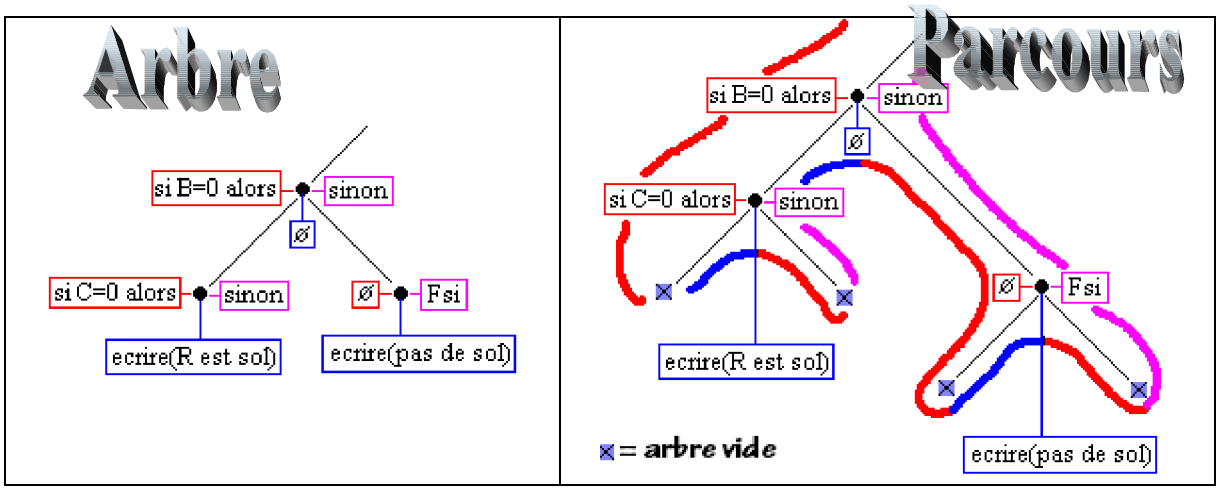
parcourir (Arbre)

si Arbre ≠ ∅ alors
  Traiter-1 (**Attribut n°1**) ;
  parcourir (Arbre.filsG) ;
  Traiter-2 (**Attribut n°2**) ;
  parcourir (Arbre.filsD) ;
  Traiter-3 (**Attribut n°3**) ;
Fsi

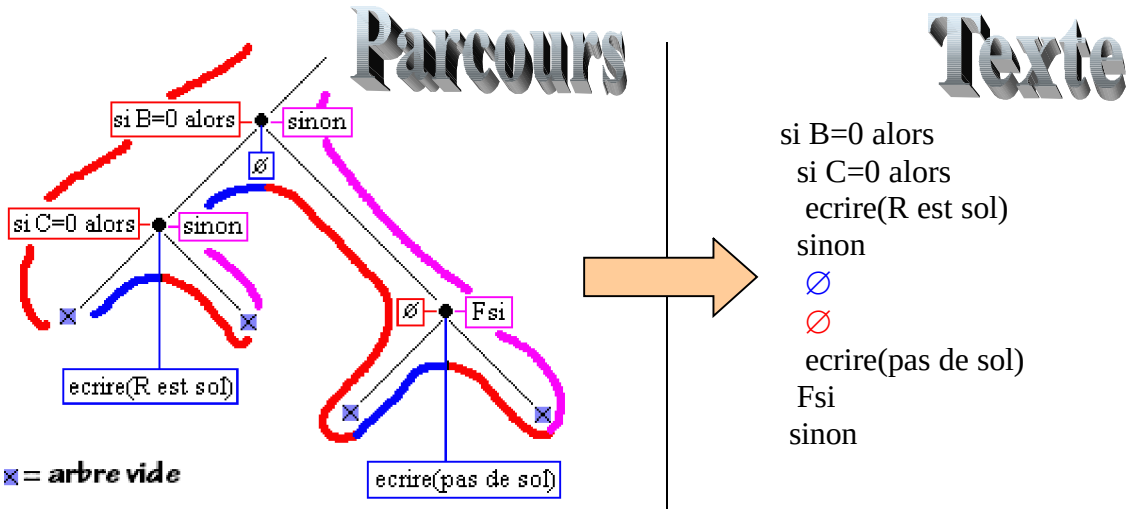


Rappelons que le symbole ∅ représente la chaîne vide il est uniquement mis dans le texe dans le but de permettre le suivi du parcours de l'arbre.

Pour bien comprendre le parcours aux feuilles de l'arbre précédent, nous avons fait figurer ci-dessous sur un exemple, les **noeuds vides** de chaque feuille et le **parcours complet associé** :

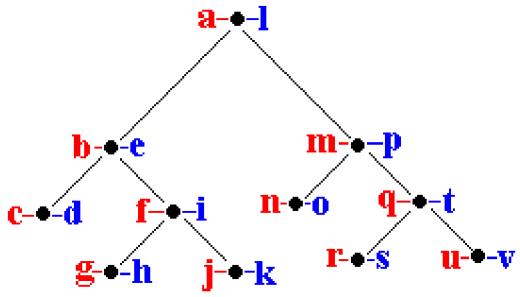


Le parcours partiel ci-haut produit le texte algorithmique suivant (le symbole ∅ est encore écrit pour la compréhension de la traversée) :



Exercice

Soit l'arbre ci-contre possédant 2 attributs par noeuds (un symbole de type caractère)



On propose le traitement en profondeur de l'arbre comme suit : L'attribut de gauche est écrit en descendant, l'attribut de droite est écrit en remontant, il n'y a pas d'attribut ni de traitement lors de l'examen du fils droit en venant du fils gauche. écrire la chaîne de caractère obtenue par le parcours ainsi défini.	**Réponse** : **abcdfghjkiemnoqrsuvtpl**

Terminons cette revue des descriptions algorithmiques des différents parcours classiques d'arbre binaire avec le parcours en largeur (*Cet algorithme nécessite l'utilisation d'une file du type Fifo dans laquelle l'on stocke les nœuds*)**.**

Algorithme de parcours en largeur

Largeur (Arbre)

si Arbre $\neq \varnothing$ alors
  ajouter racine de l'**Arbre** dans Fifo;
  tantque Fifo $\neq \varnothing$ **faire**
    prendre premier de Fifo;
    traiter premier de Fifo;
    ajouter **filsG** de premier de Fifo dans Fifo;
    ajouter **filsD** de premier de Fifo dans Fifo;
  ftant
Fsi

2.6 Insertion, suppression, recherche dans un arbre binaire de recherche

Algorithme d'insertion dans un arbre binaire de recherche

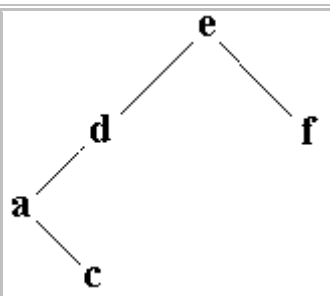
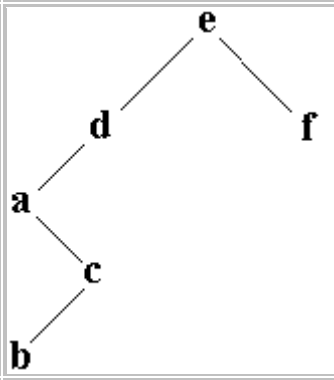
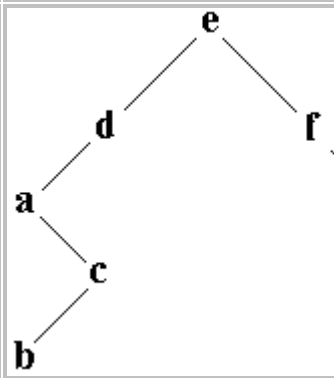
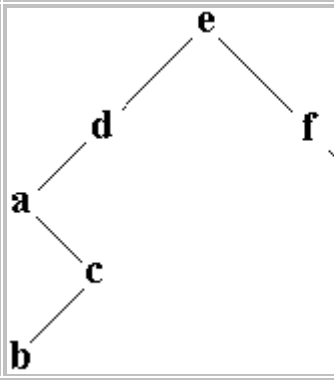
*placer l'élément Elt dans l'arbre **Arbre** par adjonctions successives aux feuilles*

placer (Arbre Elt **)**

si Arbre = ∅ **alors**
  creer un nouveau noeud contenant Elt **;**
  Arbre.Racine = ce nouveau noeud
sinon
{ - tous les éléments "info" de tous les noeuds du sous-arbre de gauche sont inférieurs ou égaux à l'élément "info" du noeud en cours (arbre)
- tous les éléments "info" de tous les noeuds du sous-arbre de droite sont supérieurs à l'élément "info" du noeud en cours (arbre)
}
  si clef (Elt) ≤ clef (**Arbre.Racine**) **alors**
    placer (Arbre.filsG Elt **)**
  sinon
    placer (Arbre.filsD Elt **)**
Fsi

Soit par exemple la liste de caractères alphabétiques : **e d f a c b u w** *, que nous rangeons dans cet ordre d'entrée dans un arbre binaire de recherche. Ci-dessous le suivi de l'algorithme de placements successifs de chaque caractère de cette liste dans un arbre de recherche:*

Insertions successives des éléments	Arbre de recherche obtenu
placer (racine , '**e**') *e est la racine de l'arbre.*	e
placer (racine , '**d**') *d < e donc fils gauche de e.*	e d
placer (racine , '**f**') *f > e donc fils droit de e.*	e d f
placer (racine , '**a**') *a < e donc à gauche, a < d donc fils gauche de d.*	e d f a

placer (racine , '**c**') *c < e donc à gauche, c < d donc à gauche, c > a donc fils droit de a.*	
placer (racine , '**b**') *b < e donc à gauche, b < d donc à gauche, b > a donc à droite de a, b < c donc fils gauche de c.*	
placer (racine , '**u**') *u > e donc à droite de e, u > f donc fils droit de f.*	
placer (racine , '**w**') *w > e donc à droite de e, w > f donc à droite de f, w > u donc fils droit de u.*	

Algorithme de recherche dans un arbre binaire de recherche

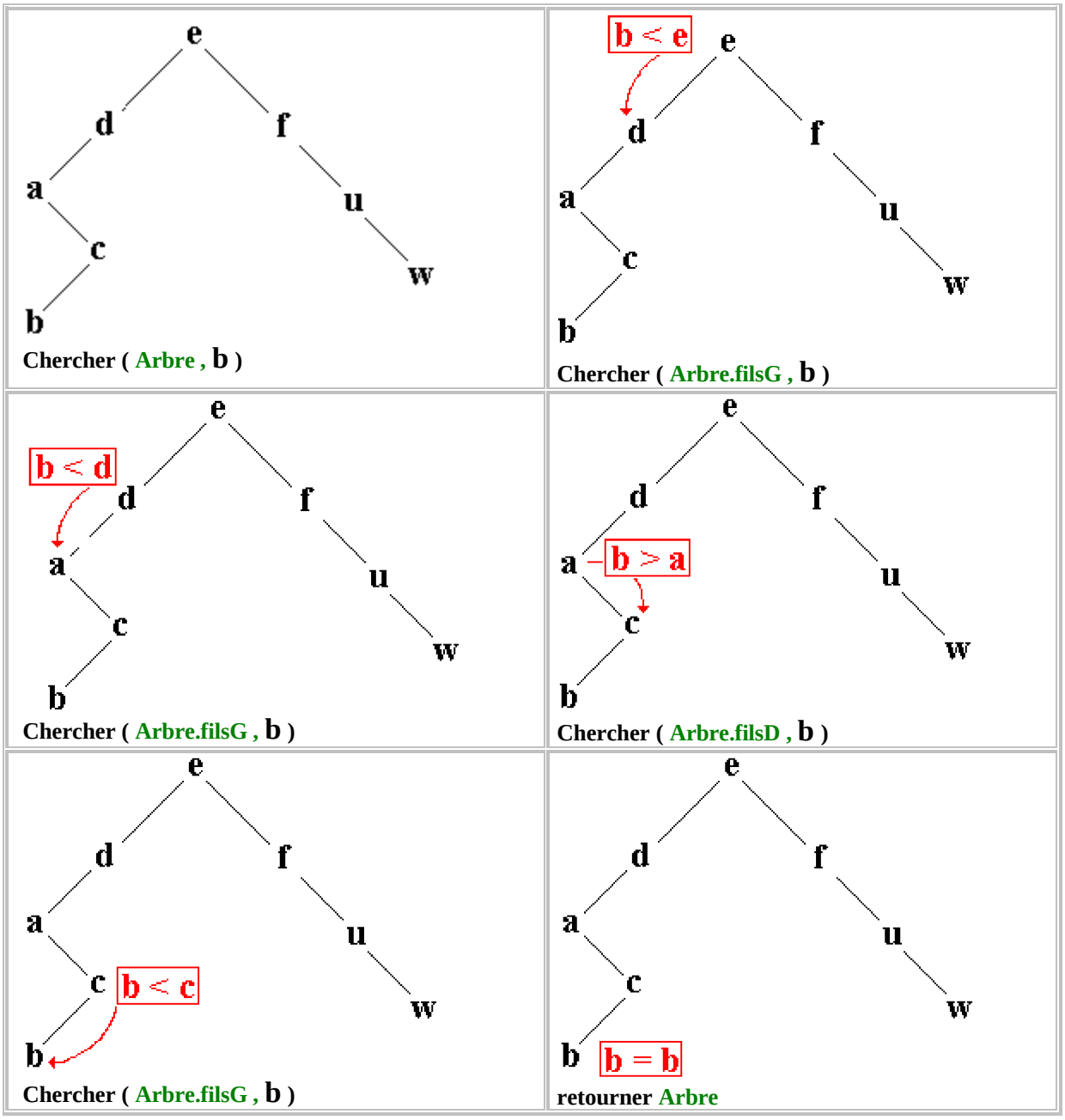
*chercher l'élément Elt dans l'arbre **Arbre :***

Chercher (Arbre Elt **) : Arbre**

si Arbre = ∅ **alors**
  Afficher Elt **non trouvé dans l'arbre;**
sinon
  si clef (Elt) < clef (**Arbre.Racine**) **alors**

```
    Chercher ( Arbre.filsG Elt ) //on cherche à gauche
  sinon
     si clef ( Elt )  >  clef ( Arbre.Racine ) alors
       Chercher ( Arbre.filsD Elt ) //on cherche à droite
     sinon retourner Arbre.Racine  //l'élément est dans ce noeud
     Fsi
  Fsi
Fsi
```

Ci-dessous le suivi de l'algorithme de recherche du caractère **b** *dans l'arbre précédent :*



Algorithme de suppression dans un arbre binaire de recherche

Afin de pouvoir supprimer un élément dans un arbre binaire de recherche, il est nécessaire de pouvoir d'abord le localiser, ensuite supprimer le noeud ainsi trouvé et éventuellement procéder à la réorganisation de l'arbre de recherche.

Nous supposons que notre arbre binaire de recherche ne possède que des éléments tous distincts (pas de redondance).

```
supprimer l'élément Elt dans l'arbre Arbre :

Supprimer ( Arbre Elt ) : Arbre
  local Node : Noeud

si Arbre = ∅ alors
  Afficher Elt non trouvé dans l'arbre;
sinon
  si clef ( Elt )  < clef ( Arbre.Racine )   alors
     Supprimer ( Arbre.filsG Elt ) //on cherche à gauche
  sinon
     si clef ( Elt )  >  clef ( Arbre.Racine ) alors
        Supprimer ( Arbre.filsD Elt ) //on cherche à droite
     sinon //l'élément est dans ce noeud
       si Arbre.filsG = ∅ alors //sous-arbre gauche vide
          Arbre ←Arbre.filsD //remplacer arbre par son sous-arbre droit
       sinon
         si Arbre.filsD = ∅ alors  //sous-arbre droit vide
          Arbre ←Arbre.filsG //remplacer arbre par son sous-arbre gauche
         sinon //le noeud a deux descendants
            Node ←PlusGrand( Arbre.filsG ); //Node = le max du fils gauche
            clef ( Arbre.Racine ) ← clef ( Node ); //remplacer etiquette
            détruire ( Node ) //on élimine ce noeud
         Fsi
       Fsi
     Fsi
   Fsi
Fsi
```

Cet algorithme utilise l'algorithme récursif **PlusGrand** *de recherche du plus grand élément dans l'arbre* ***Arbre :***

//par construction il suffit de descendre systématiquement toujours le plus à droite

```
PlusGrand ( Arbre ) : Arbre
si Arbre.filsD = ∅ alors
  retourner Arbre.Racine //c'est le plus grand élément
sinon
  PlusGrand ( Arbre.filsD  )
Fsi
```

Exercices chapitre 4

Ex-1 : On défini un nombre rationnel (fraction) comme un couple de deux entiers : le numérateur et le dénominateur. On donne ci-dessous un TAD minimal de rationnel et l'on demande de l'implanter en **Unit** Delphi.

TAD : rationnel
Utilise : **Z** *//ensemble des entiers relatifs.*
Champs : (Num , Denom) $\in$ **Z x Z***
Opérations :

Num :	———> rationnel	
Denom :	———> rationnel	
Reduire :	rationnel	———> rationnel
Addratio :	rationnel x rationnel	———> rationnel
Divratio :	rationnel x rationnel	———> rationnel
Mulratio :	rationnel x rationnel	———> rationnel
AffectQ :	rationnel	———> rationnel
OpposeQ :	rationnel	———> rationnel

Préconditions :

Divratio (x,y) **<u>defssi</u>** y.Num $\neq$ 0

Finrationnel

Ex-2 : On défini un nombre complexe à coefficient entiers relatifs comme un couple de deux entiers relatifs : la partie réelle et la partie imaginaire. On donne ci-dessous un TAD minimal de nombre et l'on demande de l'implanter en **Unit** Delphi.

TAD complexe
Utilise : **Z**
Champs : (part_reel , part_imag) $\in$ **Z x Z***
Opérations :

part_reel :	———> **Z**	
part_imag :	———> **Z**	
Charger :	**Z x Z** ———> complexe	
AffectC :	complexe	———> complexe
plus :	complexe x complexe	———> complexe
moins:	complexe x complexe	———> complexe
mult :	complexe x complexe	———> complexe
OpposeC :	complexe	———> complexe

Préconditions :

aucune

Fincomplexe

Ex-3 : On défini un nombre complexe à coefficient rationnel comme un couple de deux rationnels : la partie réelle et la partie imaginaire. On demande de construire un TAD minimal de nombre complexe utilisant le TAD rationnel et de l'implanter en **Unit** Delphi.

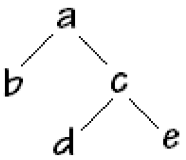
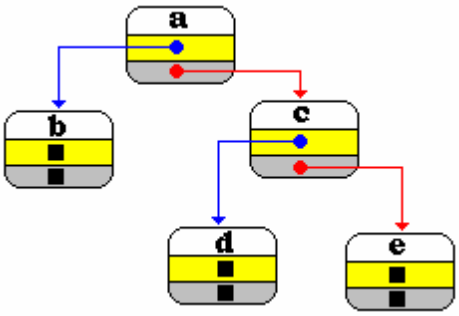
Ex-4 : En reprenant l'énoncé de l'exercice Ex-1 TAD rationnel, on implante sous forme de **classe** Delphi ce TAD et on compare son implantation en **Unit**.

Ex-5 : En reprenant l'énoncé de l'exercice Ex-3 TAD de nombre complexe à coefficients rationnels, on implante sous forme de **classe** Delphi ce TAD et on compare son implantation en **Unit**.

Ex-6 : Implantations des 4 algorithmes de parcours d'un arbre binaire étudiés dans le cours :

Il est demandé d'implanter en Delphi chacun des trois parcours en profondeur par la gauche définissant un ordre implicite (préfixé, infixé, postfixé) sur l'affichage et le traitement des données de type **string** contenues dans un arbre binaire, et le parcours en largeur avec une file Fifo.

Il est demandé d'écrire en Delphi console, l'implantation de la structure d'arbre binaire et des algorithmes de parcours avec des variables dynamiques de type **parbre = ^arbre** où arbre est un type **record** à définir selon le schéma fournit par l'exemple ci-dessous :

L'arbre :	Est implanté par les pointeurs **parbre** suivants :
	

Le traitement au nœud consistera à afficher la donnée de type string.

Puis uniquement lorsque vous aurez lu les chapitre sur les classes et la programmation objet, vous reprendrez l'implantation de la structure d'arbre binaire et des algorithmes de parcours avec une classe **TreeBin** selon le modèle ci-après :

```
TreeBin = class
    private
      FInfo : string ;
      procedure FreeRecur( x :TreeBin ) ;
      procedure PreOrdre ( x : TreeBin ; var res : string);
      procedure PostOrdre ( x : TreeBin ; var res : string);
    public
      filsG , filsD : TreeBin ;
      constructor Create(s:string; fg , fd : TreeBin);
      destructor Liberer;
      function Prefixe : string ;
      function Postfixe : string ;
      property Info:string read FInfo write FInfo;
 end;
```

Ex-7 : Il est demandé d'implanter en Delphi les 3 algorithmes d'insertion, de suppression et de recherche dans un arbre binaire de recherche ; comme dans l'exercice précédent vous proposerez une version avec variables dynamiques et une version avec une classe **TreeBinRech** héritant de la classe **TreeBin** définie à l'exercice précédent.

Ex-1 TAD rationnel - solution en Unit Delphi

Spécifications opérationnelles

TAD : rationnel
Utilise : **Z** *//ensemble des entiers relatifs.*
Champs : (Num , Denom) ∈ **Z x Z***
Opérations :

Num : ———> rationnel
Denom : ———> rationnel
Reduire : rationnel ———> rationnel
Addratio : rationnel x rationnel ———> rationnel
Divratio : rationnel x rationnel ———> rationnel
Mulratio : rationnel x rationnel ———> rationnel
AffectQ : rationnel ———> rationnel
OpposeQ : rationnel ———> rationnel

Préconditions :

Divratio (x,y) **defssi** y.Num ≠ 0

Finrationnel

Reduire : rendre le rationnel irréductible en calculant le pgcd du numérateur et du dénominateur, puis diviser les deux termes par ce pgcd.
Addratio : addition de deux nombres rationnels par la recherche du plus petit commun multiple des deux dénominateurs et mise de chacun des deux rationnels au même dénominateur.
Mulratio : multiplication de deux nombres rationnels, par le produit des deux dénominateurs et le produit des deux numérateurs.
Divratio : division de deux nombres rationnels, par le produit du premier par l'inverse du second.
AffectQ : affectation classique d'un rationnel dans un autre rationnel.
OpposeQ : renvoie l'opposé d'un rationnel dans un autre rationnel

Spécifications d'implantation

La structure de données choisie est le type record permettant de stocker les champs numérateur et dénominateur d'un nombre rationnel :

```
unit Uratio;  {unité de rationnels spécification classique ZxZ/R }
interface
    type     rationnel = record
                              num: integer;
                              denom: integer
                         end;
procedure reduire (var r: rationnel);
procedure addratio (a, b: rationnel; var s: rationnel);
procedure divratio (a, b: rationnel; var s: rationnel);
procedure mulratio (a, b: rationnel; var s: rationnel);
procedure affectQ(var s: rationnel; b: rationnel);
procedure opposeQ(x:rationnel;var s:rationnel);
```

Code de la Unit : Uratio

```
unit Uratio; {unité de rationnels spécification classique ZxZ/R}

interface

 type
  rationnel =
  record
   num: integer;
   denom: integer
  end;
  procedure reduire (var r: rationnel);
  procedure addratio (a, b: rationnel; var s: rationnel);
  procedure divratio (a, b: rationnel; var s: rationnel);
  procedure mulratio (a, b: rationnel; var s: rationnel);
  procedure affectQ(var s: rationnel; b: rationnel);
  procedure opposeQ(x:rationnel;var s:rationnel);

implementation

procedure maxswap (var a: integer; var b: integer);
var
   t: integer;
   begin
     if a < b then
     begin
       t := a;
       a := b;
       b := t
     end;
   end;
```

```
---------------- SPECIFICATIONS ---------------
maxswap : N x N ---> N x N
met le plus grand des deux entiers a et b dans a,
et le plus petit dans b.
local: t
paramètre-Entrée: a, b
paramètre-Sortie: a, b
```

```
function pgcd (a, b: integer): integer;
  var
   r: integer;
   begin
     maxswap(a, b);
     if a*b=0 then
       pgcd:=1
     else
     begin
       repeat
         r := a mod b;
         a := b;
         b := r
       until
       r = 0;
       pgcd := a
     end
   end;
```

```
---------------- SPECIFICATIONS ---------------
pgcd : N° x N° ---> N
renvoie le pgcd de deux entiers non nuls
pgcd=1, si l'un des 2 au moins est nul.
local: r
paramètre-Entrée: a, b
paramètre-Sortie: pgcd
utilise: maxswap
```

```
function ppcm (a, b: integer): integer;
 var
  k, p: integer;
  begin
   maxswap(a, b);
   if a*b=0 then
    ppcm:=0
   else
   begin
    k := 1;
    p := b;
    while (k <= a) and (p mod a <> 0) do
    begin
     p := b * k;
     k := k + 1
    end;
    ppcm := p
   end
  end;
```

```
---------------- SPECIFICATIONS ---------------
ppcm : N° x N° ---> N
renvoie le ppcm de deux entiers non nuls
renvoie 0, si l'un des 2 au moins est nul.
local: k, p
paramètre-Entrée: a, b
paramètre-Sortie: ppcm
utilise: maxswap
```

```
procedure reduire (var r: rationnel);
 var    pg: integer;
  begin
   if r.denom=0 then
    halt
   else
   begin
    pg := pgcd(r.num, r.denom);
    if pg <> 1 then
    begin
     r.num := r.num div pg;
     r.denom := r.denom div pg
    end;
    if (r.num>0) and (r.denom>0)
     or (r.num<0) and (r.denom<0) then
    begin {positif}
     r.num:=abs(r.num);
     r.denom:=abs(r.denom);
    end
    else
    begin{négatif}
     r.num:=-abs(r.num);
     r.denom:=abs(r.denom);
    end
   end
  end;
```

```
---------------- SPECIFICATIONS ---------------
reduire : Q ---> Q
rend un rationnel non nul irréductible
arrête l'exécution, si le dénominateur est nul.
Le signe est détenu par le numérateur.
local: pg
paramètre-Entrée: r
paramètre-Sortie: r
utilise: pgcd
```

```
procedure affectQ( var s: rationnel; b: rationnel);
 begin
  s.num:=b.num;
  s.denom:=b.denom
 end;
```

```
procedure opposeQ(x:rationnel;var s:rationnel);
 begin
  s.num:=-x.num;
  s.denom:=x.denom
 end;
```

```
procedure addratio (a, b: rationnel; var s: rationnel);
 var
  divcom, coeff_a, coeff_b: integer;
  begin
   reduire(a);
   reduire(b);
   divcom := ppcm(a.denom, b.denom);
   coeff_a := divcom div a.denom;
   coeff_b := divcom div b.denom;
   s.num := a.num * coeff_a + b.num * coeff_b;
   s.denom := divcom;
   reduire(s);
  end;
```

```
---------------- SPECIFICATIONS ---------------
addratio : Q x Q ---> Q
donne dans s la somme des deux rationnels non
nuls, s=a+b
a et b sont rendus irréductibles avant le calcul.
Le résultat s est rendu irréductible après calcul.
local: divcom, coeff_a, coeff_b
paramètre-Entrée: a, b
paramètre-Sortie: s
utilise: reduire,ppcm
```

```
procedure divratio (a, b: rationnel; var s: rationnel);
 begin
  reduire(a);
  reduire(b);
  if b.num=0 then
   halt
  else
  begin
   s.num := a.num * b.denom;
   s.denom := a.denom * b.num;
   reduire(s)
  end
 end;
```

```
---------------- SPECIFICATIONS ---------------
divratio : Q x Q ---> Q
donne dans s le rapport des deux rationnels non
nuls, s=a/b
a et b sont rendus irréductibles avant le calcul.
Le résultat s est rendu irréductible après calcul.
paramètre-Entrée: a, b
paramètre-Sortie: s
utilise: reduire
```

```
procedure mulratio (a, b: rationnel; var s: rationnel);
 begin
  reduire(a);
  reduire(b);
  s.num := a.num * b.num;
  s.denom := a.denom * b.denom;
  reduire(s)
 end;

end.
```

```
---------------- SPECIFICATIONS ---------------
multratio : Q x Q ---> Q
donne dans s le produit des deux rationnels non
nuls, s=a*b
a et b sont rendus irréductibles avant le calcul.
Le résultat s est rendu irréductible après calcul.
paramètre-Entrée: a, b
paramètre-Sortie: s
utilise: reduire
```

Utilisation de la unit

```
program essaiRatio;  {début programme de test de la unit Uratio de nombres rationnels   }
 uses Uratio;
 var    r1, r2, r3, r4, r5 : rationnel;
  begin     { exemple de calcul sur les rationnels non nuls à continuer}
   r1.num :=18;      r1.denom := 15;      r2.num := 7;      r2.denom := 12;
   addratio(r1, r2, r3);
   writeln('18/15 + 7/12 = ', r3.num, '/', r3.denom);
   mulratio(r1, r2, r4);
   writeln('18/15 * 7/12 = ', r4.num, '/', r4.denom);  …..
```

Ex-2 TAD nombre complexe à coeff entiers - solution en Unit Delphi

Spécifications opérationnelles

TAD complexe
Utilise : **Z**
Champs : (part_reel , part_imag) ∈ **Z x Z***
Opérations :

part_reel :	———> **Z**
part_imag :	———> **Z**
Charger :	**Z x Z** ———> complexe
AffectC :	complexe ———> complexe
plus :	complexe x complexe ———> complexe
moins:	complexe x complexe ———> complexe
mult :	complexe x complexe ———> complexe
OpposeC :	complexe ———> complexe

Préconditions :
aucune
Fincomplexe

Charger : remplit les deux champs part_reel et part_imag d'un nombre complexe.
AffectC : affectation classique d'un complexe dans un autre.
plus : addition de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x+x')+(y+y')i.
moins : soustraction de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1-z_2=(x-x')+(y-y')i.
mult : multiplication de 2 nombres complexes spécif mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x.x'- y.y')+(x.y'+x'.y)i.
OpposeC : pour un nombre complexe x+iy cet opérateur renvoie –x –iy

Spécifications d'implantation

La structure de données choisie est aussi ici le type **record** permettant de stocker les champs partie réelle et partie imaginaire d'un nombre complexe :

```
unit UComplx1; {unit de calcul sur les nombres complexes à coefficients entiers}
interface
  type    complex = record
                        part_reel: integer;
                        part_imag: integer
                    end;

  procedure Charger (x, y: integer; var z: complex);
  procedure affectC (var z: complex;y: complex );
  procedure plus (x, y: complex; var z: complex);
  procedure moins (x, y: complex; var z: complex);
  procedure mult (x, y: complex; var z: complex);
  procedure OpposeC (x:complex; var z:complex);
```

Code de la Unit : UComplx1

```
unit UComplx1;{unit de calcul sur les nombres complexes à coefficients entiers }

interface

 type
  complex =    record
   part_reel: integer;
   part_imag: integer
  end;

  procedure Charger (x, y: integer; var z: complex);
  procedure affectC (var z: complex;y: complex );
  procedure plus (x, y: complex; var z: complex);
  procedure moins (x, y: complex; var z: complex);
  procedure mult (x, y: complex; var z: complex);
  procedure OpposeC(x:complex;var z:complex);

implementation

procedure Charger (x, y: integer; var z: complex);
  begin
   z.part_reel := x;
   z.part_imag := y
  end;

  procedure affectC (var z: complex;y: complex );
  begin
   z.part_reel := y.part_reel;
   z.part_imag := y.part_imag
  end;

  procedure plus (x, y: complex; var z: complex);
  begin
   z.part_reel := x.part_reel + y.part_reel;
   z.part_imag := x.part_imag + y.part_imag
  end;

procedure moins (x, y: complex; var z: complex);
begin
  z.part_reel := x.part_reel - y.part_reel;
   z.part_imag := x.part_imag - y.part_imag
  end;

  procedure mult (x, y: complex; var z: complex);
  begin
   z.part_reel := x.part_reel * y.part_reel
                    - x.part_imag * y.part_imag;
   z.part_imag := x.part_reel * y.part_imag
                   + y.part_reel * x.part_imag
  end;

  procedure OpposeC(x:complex;var z:complex);
  begin
   z.part_reel := -x.part_reel;
   z.part_imag := x.part_imag
  end;

 end.
```

Utilisation de la unit

```
program essai_complexe1;
 {test de l'utilisation de la unit Ucomplx1
  de nombres complexes à coeff. Entiers   }
 uses
  Ucomplx1;

 var
  u, z, z1, z2: complex;

procedure ecrit (v: string; z: complex);
begin
   writeln(v, ': ', z.part_reel : 3,
                '+ ', z.part_imag : 3, '.i')
end;

begin
    writeln('_________');
    Charger(2, -3, z1);
    ecrit('z1', z1);
    Charger(8, 113, z2);
    ecrit('z2', z2);
    affectC(z, z1);
    ecrit('z', z);
    plus(z1, z2, u);
    ecrit('u=z1+z2', u);
    moins(z1, z2, u);
    ecrit('u=z1-z2', u);
    Charger(1, 0, z);
    mult(z1, z2, u);
    ecrit('u=z1*z2', u);
    mult(z, z2, u);
    ecrit('u=1*z2', u);
end.
```

Le TAD complexe utilisant Le TAD rationnel

TAD complexe
Utilise : **rationnel**
Champs : (part_reel , part_imag) ∈ **rationnel x rationnel**
Opérations :

part_reel :	———> **rationnel**
part_imag :	———> **rationnel**
Charger :	**rationnel x rationnel** ———> complexe
AffectC :	complexe ———> complexe
plus :	complexe x complexe ———> complexe
moins:	complexe x complexe ———> complexe
mult :	complexe x complexe ———> complexe
OpposeC :	complexe ———> complexe

Préconditions :
aucune
Fincomplexe

Spécifications opérationnelles

(identiques à l'exercice précédent)

Charger : remplit les deux champs part_reel et part_imag d'un nombre complexe.
AffectC : affectation classique d'un complexe dans un autre.
plus : addition de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x+x')+(y+y')i.
moins : soustraction de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1-z_2=(x-x')+(y-y')i.
mult : multiplication de 2 nombres complexes spécif mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x.x'- y.y')+(x.y'+x'.y)i.
OpposeC : pour un nombre complexe x+iy cet opérateur renvoie –x –iy

Spécifications d'implantation

```
unit UComplx2; {unit de nombres complexes à
  coefficients rationnels utilisant
  la unit de rationnels déjà construite Uratio }

interface
uses Uratio;
  type
    complex = record
        part_reel: rationnel;
        part_imag: rationnel
      end;
```

```
procedure Charger (x, y: integer; var z: complex);
procedure affectC (var z: complex;y: complex );
procedure plus (x, y: complex; var z: complex);
procedure moins (x, y: complex; var z: complex);
procedure mult (x, y: complex; var z: complex);
procedure OpposeC(x:complex; var z:complex);
```

Code de la Unit : Ucomplx2

```
unit UComplx2;
{unit de nombres complexes à coefficients rationnels utilisant la unit de rationnels déjà construite Uratio }
interface
 uses Uratio;
 type
  complex =    record
    part_reel: rationnel;
    part_imag: rationnel
  end;

  procedure Charger (x, y: rationnel; var z: complex);
  procedure affectC (var z: complex;y: complex );
  procedure plus (x, y: complex; var z: complex);
  procedure moins (x, y: complex; var z: complex);
  procedure mult (x, y: complex; var z: complex);
  procedure OpposeC(x:complex;var z:complex);

implementation

procedure OpposeC ( x:complex; var z:complex); (* fournit l'opposé de x dans z *)
  begin
    OpposeQ(x.part_reel,z.part_reel);
    OpposeQ(x.part_imag,z.part_imag);
  end;

 procedure Charger (x, y : rationnel; var z : complex);
 begin
  reduire(x);
  reduire(y);
  affectQ(z.part_reel,x);
  affectQ(z.part_imag,y);
 end;

 procedure affectC (var z: complex;y: complex );
 begin
  Charger(y.part_reel,y.part_imag,z)
 end;

 procedure plus (x, y: complex; var z: complex);
 var r1,r2:rationnel;
 begin
  addratio(x.part_reel,y.part_reel,r1);
  addratio(x.part_imag, y.part_imag,r2);
  Charger (r1,r2,z)
 end;

 procedure moins (x, y: complex; var z: complex);
 var z1:complex;
 begin
  OpposeC(y,z1);
  plus(x,z1,z)
 end;

 procedure mult (x, y: complex; var z: complex);
 var r1,r2,r3,r4:rationnel;
 begin
  {z.part_reel := x.part_reel * y.part_reel - x.part_imag * y.part_imag;}
```

```
  mulratio(x.part_reel,y.part_reel,r1);
  mulratio(x.part_imag, y.part_imag,r2);
  OpposeQ(r2,r3);
  addratio(r1,r3,r4);

  {z.part_imag := x.part_reel * y.part_imag + y.part_reel * x.part_imag}
  mulratio(x.part_reel,y.part_imag,r1);
  mulratio(x.part_imag, y.part_reel,r2);
  addratio(r1,r2,r3);
  Charger(r4,r3,z)
 end;

end .
```

Utilisation de la unit Ucomplx2

```
program essai_complexe2;
 {programme de test et d'utilisation de la unit Ucomplx2  de nombres complexes à coeff. rationnels  }
 uses
  Uratio,Ucomplx2 ;
 procedure ecrit (v: string; z: complex);
 begin
  writeln(v, ': (', z.part_reel.num,'/', z.part_reel.denom, ')+ (', z.part_imag.num,'/', z.part_imag.denom, ').i')
 end;
 var    r1,r2,r3:rationnel;
        u, z, z1, z2: complex;
  begin
   writeln('_______________________');
   {//// les chargements dépendent du type des données ////}
   r1.num :=18;
   r1.denom := 15;
   r2.num := 7;
   r2.denom := 12;
   Charger(r1, r2, z1);
   ecrit('z1', z1);
   r1.num :=35;
   r1.denom := 25;
   r2.num := 11;
   r2.denom := 6;
   Charger(r1, r2, z2);
   ecrit('z2', z2);
   {//// les appels d'opérateurs sont identiques à ceux de l'exercice précédent: ////}
   affectC(z, z1);
   ecrit('z=z1', z);
   plus(z1, z2, u);
   ecrit('u=z1+z2', u);
   moins(z1, z2, u);
   ecrit('u=z1-z2', u);
   r1.num :=1;
   r1.denom := 1;
   r2.num := 0;
   r2.denom := 1;
   Charger(r1, r2, z1);
   mult(z1, z2, u);
   ecrit('u=1*z2', u);
   mult(z, z2, u);
   ecrit('u=z1*z2', u);
  end.
```

Ex-4 TAD nombre rationnel - solution en classe Delphi

Le TAD	La classe
TAD : rationnel **Utilise** : **Z** *//ensemble des entiers relatifs.* **Champs** : (Num , Denom) ∈ **Z x Z*** **Opérations** : Num : ———> rationnel Denom : ———> rationnel Reduire : rationnel ———> rationnel Addratio : rationnel × rationnel ———> rationnel Divratio : rationnel × rationnel ———> rationnel Mulratio : rationnel × rationnel ———> rationnel AffectQ : rationnel ———> rationnel OpposeQ : rationnel ———> rationnel **Préconditions** : Divratio (x,y) **defssi** y.Num ≠ **Finrationnel**	rationnel denom : integer num : integer maxswap(...) pgcd(...) ppcm(...) Addratio(...) affectQ(...) Divratio(...) Mulratio(...) opposeQ(...) reduire(...)

En Delphi

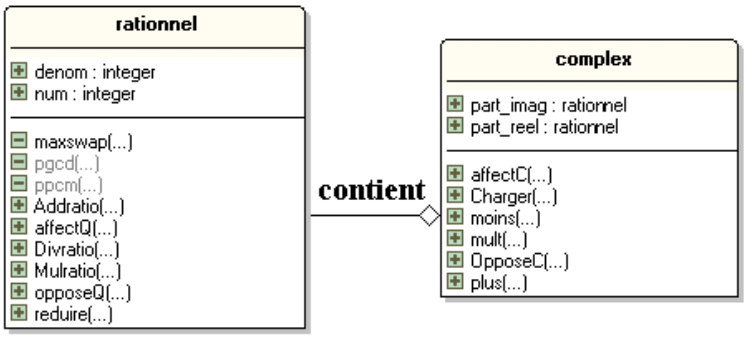
Classe

```
unit UClasseratio;
 interface
 type
  rationnel = class
   private
      function pgcd (a, b: integer): integer;
      function ppcm (a, b: integer): integer;
      procedure maxswap (var a: integer;
                         var b: integer);
   public
      num: integer;
      denom: integer;
      procedure reduire ;
      procedure Addratio (a,s: rationnel);
      procedure Divratio (a,d: rationnel);
      procedure Mulratio (a,m: rationnel);
      procedure affectQ (s: rationnel);
      procedure opposeQ (s:rationnel);
  end;

implementation
 procedure rationnel.maxswap (var a: integer;
                              var b: integer); …
 function rationnel.pgcd (a, b: integer): integer; …
 function rationnel.ppcm (a, b: integer): integer; …
 procedure rationnel.reduire ; …
 procedure rationnel.Addratio (a,s: rationnel); …
 procedure rationnel.Divratio (a,d: rationnel); …
 procedure rationnel.Mulratio (a,m: rationnel); …
 procedure rationnel.affectQ (s: rationnel); …
 procedure rationnel.opposeQ (s: rationnel); …
```

Unit

```
unit Unitratio;
interface
 type
  rationnel = record
   num: integer;
   denom: integer
  end;

  procedure reduire (var r: rationnel);
  procedure Addratio (a, b: rationnel; var s: rationnel);
  procedure Divratio (a, b: rationnel; var s: rationnel);
  procedure Mulratio (a, b: rationnel; var s: rationnel);
  procedure affectQ (var s: rationnel; b: rationnel);
  procedure opposeQ (x:rationnel;var s:rationnel);

implementation

 procedure maxswap (var a: integer; var b: integer); …
 function pgcd (a, b: integer): integer; …
 function ppcm (a, b: integer): integer; …
 procedure reduire (var r: rationnel); …
 procedure Addratio (a, b: rationnel; var s: rationnel); …
 procedure Divratio (a, b: rationnel; var s: rationnel); …
 procedure Mulratio (a, b: rationnel; var s: rationnel); …
 procedure affectQ (var s: rationnel; b: rationnel); …
 procedure opposeQ (x: rationnel;var s: rationnel); …
```

Ex-5 TAD nombre complexe - solution en classe Delphi

Le TAD	La classe complex utilise la classe rationnel
TAD complex **Utilise** : **Z** **Champs** : (part_reel , part_imag) ∈ **Z x Z*** **Opérations** : part_reel : → **Z** part_imag : → **Z** Charger : **Z** × **Z** → complex AffectC : complex → complex plus : complex × complex → complex moins: complex × complex → complex mult : complex × complex → complex OpposeC : complex → complex **Fincomplexe**	

En Delphi

Classe

```
unit UClasseComplexe2;

 interface
 uses UClasseratio;
 type
   complex = class
 private
 //pas d'éléments privés pour l'instant
 public
   part_reel: rationnel;
   part_imag: rationnel;
   procedure Charger (x, y: rationnel);
   procedure affectC (z: complex);
   procedure plus (x,z: complex);
   procedure moins (x,z: complex);
   procedure mult (x,z: complex);
   procedure OpposeC(z: complex);
end;

implementation
 procedure complex.Charger (x, y: rationnel); …
 procedure complex.affectC (z: complex); …
 procedure complex.plus (x,z:complex); …
 procedure complex.moins (x,z: complex); …
 procedure complex.mult (x,z: complex); …
 procedure complex.OpposeC(z:complex); …
end.
```

Unit

```
unit Ucomplx2;

interface
 uses Uratio;
 type
   complex =
   record
     part_reel: rationnel;
     part_imag: rationnel
   end;

   procedure Charger (x, y: rationnel; var z: complex);
   procedure affectC (var z: complex;y: complex );
   procedure plus (x, y: complex; var z: complex);
   procedure moins (x, y: complex; var z: complex);
   procedure mult (x, y: complex; var z: complex);
   procedure OpposeC(x:complex;var z:complex);

implementation
 procedure Charger (x, y: rationnel; var z: complex); …
 procedure affectC (var z: complex;y: complex ); …
 procedure plus (x, y: complex; var z: complex); …
 procedure moins (x, y: complex; var z: complex); …
 procedure mult (x, y: complex; var z: complex); …
 procedure OpposeC(x:complex;var z:complex); …
end.
```

Ex-6 : Solution des 4 algorithmes de parcours d'un arbre

*Nous implantons en Delphi les 4 algorithmes dont 3 sont sous forme de procédures récursives. Nous supposons que les informations stockées dans un noeud sont du type chaîne de caractère (**string**), le traitement consistera ici à écrire le contenu de la string d'un noeud lorsqu'il est parcouru. La structure de données d'arbre est représentée par*

Implantations des données avec variables dynamique et classe

Nous proposons parallèlement les deux implantations demandées que le lecteur testera sur sa machine en fonction de son avancement dans le cours.

Implantation en Delphi avec des variables dynamiques :

```
type
   parbre = ^arbre;
   arbre = record
              info : string ;
              filsG, filsD: parbre
         end;
```

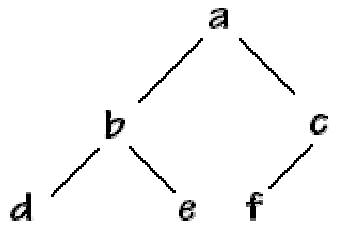
Implantation en Delphi avec une classe :

```
interface
// dans cette classe tous les champ sont publics afin de simplifier l'écriture
type
  TreeBin = class
     private
        procedure FreeRecur( x :TreeBin ) ;
     public
        Info : string;
        filsG , filsD : TreeBin;
        constructor CreerTreeBin(s:string);overload;
        constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
        destructor Liberer;
   end;
implementation
{-------------------   Méthodes privé -------------------------}
procedure TreeBin.FreeRecur ( x : TreeBin ) ;
 // parcours postfixe pour détruire les objets de l'arbre x
begin
  FreeRecur( x.filsG );
  FreeRecur( x.filsD );
  x.Free
end;

{-------------------   Méthodes public -------------------------}
constructor TreeBin.CreerTreeBin(s:string);
 begin
    self.info := s;
    self.filsG := nil;
    self.filsD := nil;
 end;

 constructor TreeBin.CreerTreeBin(s : string ; fg, fd : TreeBin);
 begin
    self.info := s;
    self.filsG := fg;
```

```
    self.filsD := fd;
 end;

destructor TreeBin.Liberer;
 // la destruction de tout l'arbre :
begin
  FreeRecur (self) ;
  self := nil;
end;

end.
```

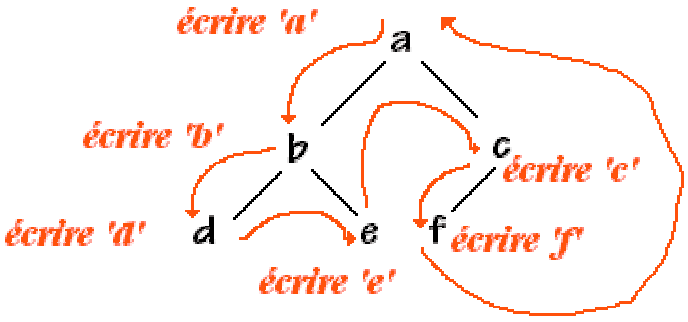
Nous prenons comme exemple sur lequel appliquer les 4 algorithmes, l'arbre binaire X suivant (chaque noeud a une info de type caractère stocké dans une string) :

X =



Implantation de l'algorithme de parcours en pré-ordre :

```
parcourir ( Arbre )
si Arbre ≠∅  alors
    Traiter-1 (info(Arbre.Racine)) ;
    parcourir ( Arbre.filsG ) ;
    parcourir ( Arbre.filsD ) ;
Fsi
```



La procédure écrira successivement : **abdecf**

Préordre en Delphi avec les variables dynamiques :

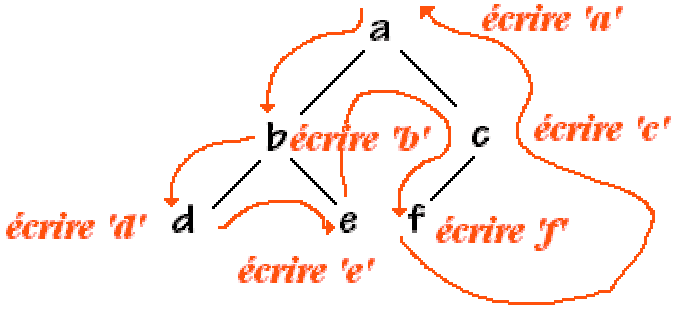
```
procedure prefixe (f : parbre);
 begin
  if f <> nil then
   with f^ do
    begin
      write(info);
      prefixe( filsG );
      prefixe( filsD )
    end
 end;
```

Préordre en Delphi avec la classe :

```
interface
type
  TreeBin = class
    private
      procedure FreeRecur( x :TreeBin ) ;
      procedure PreOrdre ( x : TreeBin);
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
      procedure Prefixe;
   end;

implementation
{-------------------   Méthodes privé -------------------------}
procedure TreeBin.PreOrdre ( x : TreeBin ) ;
 // parcours préfixé d'un arbre x
begin
 if x<>nil then
 begin
   write(x.info);
   PreOrdre( x.filsG );
   PreOrdre( x.filsD )
 end
end;
//autres méthodes ......

{-------------------   Méthodes public -------------------------}
procedure Prefixe;
// parcours préfixé de l'objet
begin
  PreOrdre( self );
end;
//autres méthodes ......

end.
```

Implantation de l'algorithme en post-ordre :

```
parcourir ( Arbre )
si Arbre ≠ ∅  alors
   parcourir ( Arbre.filsG ) ;
   parcourir ( Arbre.filsD ) ;
   Traiter-3 (info(Arbre.Racine)) ;
Fsi
```



La procédure écrira successivement: **debfca**

Postordre en Delphi avec les variables dynamiques :

```
procedure postfixe (f : parbre);
 begin
  if f <> nil then
    with f^ do
      begin
        postfixe( filsG );
        postfixe( filsD );
        write(info)
      end
 end;
```

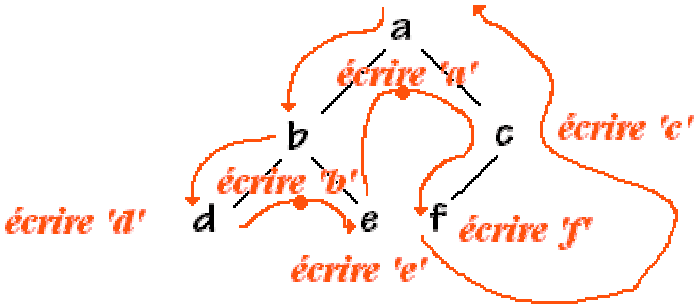
Postordre en Delphi avec la classe :

```
interface
type
  TreeBin = class
    private
      procedure FreeRecur( x :TreeBin ) ;
      procedure PreOrdre ( x : TreeBin);
      procedure PostOrdre ( x : TreeBin);
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
      procedure Prefixe;
      procedure Postfixe;
   end;
implementation
{-------------------   Méthodes privé -------------------------}
procedure TreeBin.PostOrdre ( x : TreeBin ) ;
 // parcours postfixé d'un arbre x
begin
 if x<>nil then
 begin   PostOrdre( x.filsG );
   PostOrdre( x.filsD );
   write(x.info);
 end
end;
//autres méthodes ......

{-------------------   Méthodes public -------------------------}
procedure Postfixe;
// parcours préfixé de l'objet
begin
  PostOrdre( self );
end;
//autres méthodes ......

end.
```

Implantation de l'algorithme en ordre symétrique :

```
parcourir ( Arbre )
```

```
si Arbre ≠∅  alors
   parcourir ( Arbre.filsG ) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```



La procédure écrira successivement : **dbeafc**

Ordre infixé en Delphi avec les variables dynamiques :

```
procedure infixe (f : parbre);
  begin
    if f <> nil then
      with f^ do
        begin
          infixe( filsG );
          write(info);
          infixe( filsD )
        end
  end;
```

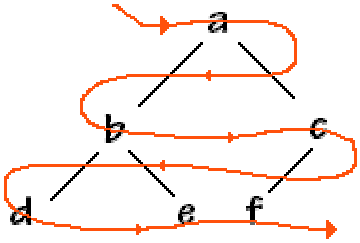
Ordre infixé en Delphi avec la classe :

```
interface
type
   TreeBin = class
      private
         procedure FreeRecur( x :TreeBin ) ;
         procedure PreOrdre ( x : TreeBin);
         procedure PostOrdre ( x : TreeBin);
         procedure InfixeOrdre ( x : TreeBin);
      public
         Info : string;
         filsG , filsD : TreeBin;
         constructor CreerTreeBin(s:string);overload;
         constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
         destructor Liberer;
         procedure Prefixe;
         procedure Postfixe;
         procedure Infixe;
    end;
implementation
{-------------------   Méthodes privé -------------------------}
procedure TreeBin.InfixeOrdre ( x : TreeBin ) ;
 // parcours infixé d'un arbre x
begin
 if x<>nil then
 begin
```

```
   InfixeOrdre( x.filsG );
  write(x.info);
  InfixeOrdre( x.filsD );
 end
end;
//autres méthodes ......

{-------------------   Méthodes public ------------------------}
procedure Infixe;
// parcours préfixé de l'objet
begin
  InfixeOrdre( self );
end;
//autres méthodes ......

end.
```

Algorithme de parcours en largeur (hiérarchique)

```
Largeur ( Arbre )
si Arbre ≠ ∅  alors
   ajouter Arbre.racine dans Fifo;
   tantque Fifo ≠ ∅ faire
     prendre premier de Fifo;
     traiter premier de Fifo;
     ajouter filsG de premier de Fifo dans Fifo;
     ajouter filsD de premier de Fifo dans Fifo;
   ftant
Fsi
```



La procédure écrira successivement : **abcdef**

Implantation en Delphi avec les variables dynamiques et un TList :

```
type
   parbre = ^arbre;
   arbre = record
              info : string ;
              filsG, filsD: parbre
         end;
```

Nous utilisons un objet Delphi de classe TList pour implanter notre Fifo.

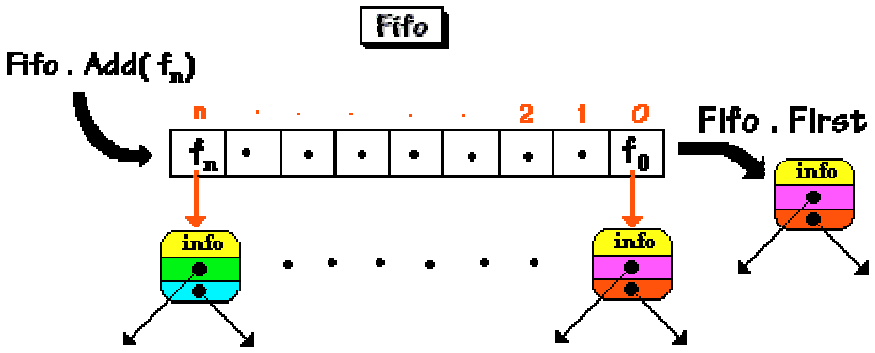
Un **TList** stocke un tableau de pointeurs (Pointer en Delphi). Un objet **TList** est souvent utilisé pour gérer une liste d'objets. **TList** introduit des propriétés et méthodes permettant d'ajouter ou de supprimer des objets de la liste. En particulier afin d'implanter une file de type Fifo, nous utiliserons les membres suivants du **TList**.

méthodes

function Add(x : Pointer): Integer; *Ajoute un nouvel élément en fin de liste et renvoie son rang.*
function First : Pointer; *Renvoie le premier élément du tableau (celui de rang 0).*

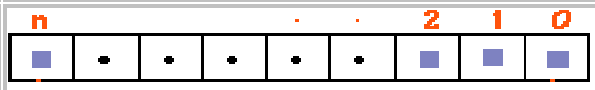
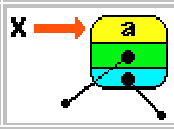
```
procedure Delete( Index : Integer); Supprime l'élément d'indice spécifié par le paramètre Index.

attributs
 property Count : Integer; Indique le nombre d'entrées utilisées de la liste.
```

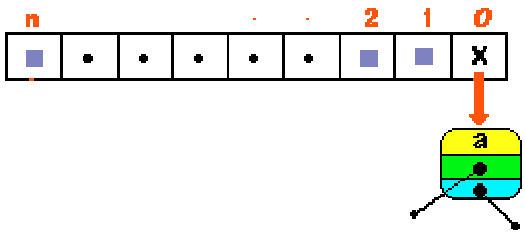
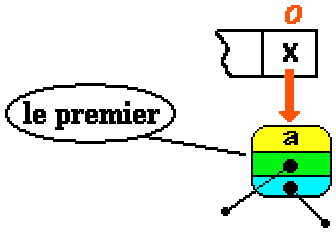
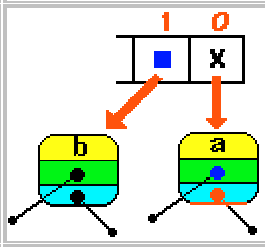
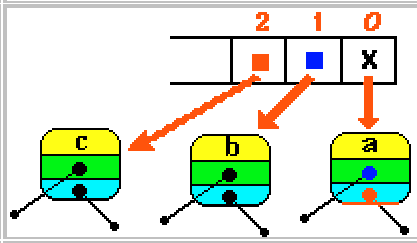
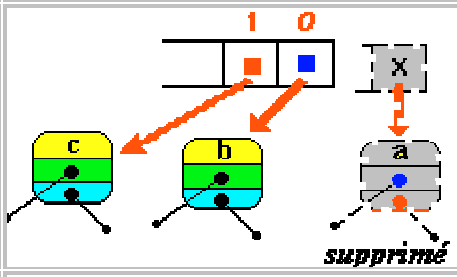
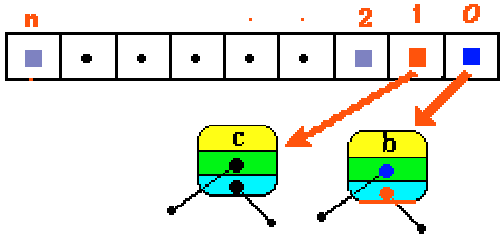
Le **TList** ne contiendra pas les noeuds de l'arbre eux-mêmes mais les pointeurs vers les noeuds (type **parbre** ici) :




```
procedure Largeur ( x : parbre);
 var Fifo : TList;
 begin
  if f <> nil then
  begin
   Fifo:=TList.Create;  // crée la Fifo
   Fifo.Add( x ); // ajoute la racine x dans Fifo
   while Fifo.Count<>0 do
   begin
    write(parbre(Fifo.First)^.info); // traitement du premier
    if parbre(Fifo.First)^.filsG <> nil then
        Fifo.Add(parbre(Fifo.First)^.filsG); // ajoute le fils gauche du premier dans Fifo
    if parbre(Fifo.First)^.filsD <> nil then
        Fifo.Add(parbre(Fifo.First)^.filsD); // ajoute le fils droit du premier dans Fifo
    Fifo.delete(0); // supprime l'élément de rang 0 (le premier)
   end;
   Fifo.Free ; // supprime la Fifo
  end
 end;
```

On applique la procédure Largeur à l'arbre X afin de parcourir et d'écrire hiérarchiquement les caractères de chaque noeud. Ci-dessous le début d'un suivi d'exécution de la procédure Largeur :

Instruction exécutée	Action sur le TList Fifo
Fifo:=TList.Create;	
// ajouter la racine x dans Fifo Fifo.Add(x);	

	
Début de la boucle while : *// traitement du premier* write(parbre(Fifo.First)^.info)	 ecrit : **a**
// ajoute le fils gauche du premier dans Fifo Fifo.Add(parbre(Fifo.First)^.filsG)	
// ajoute le fils droit du premier dans Fifo Fifo.Add(parbre(Fifo.First)^.filsD)	
// supprime l'élément de rang 0 (le premier) Fifo.delete(0);	
Reprise de la boucle while : *// traitement du premier* write(parbre(Fifo.First)^.info) etc	 ecrit : **b**

Programmes Delphi complets

Ci-dessous un programme complet en Delphi (avec variables dynamiques) à exécuter tel quel avec Delphi en mode console :

```
program Tree;
 {les arbres sont parcourus sous les 3 formes
  in, post et pré-fixées
 }
 {$APPTYPE CONSOLE}
 uses   sysutils;
 type
  parbre = ^arbre;
  arbre =
  record
   val: char;
   g, d: parbre
  end;
  var
   rac,arb1,arb2,arb3,arb4,arb5: parbre;
```

```
procedure construit (var rac:parbre;
filsg,filsd:parbre;elt:string);
 // construit un arbre
 begin
  if rac=nil then
  begin
   new(rac);
   with rac^ do
   begin
    val := elt;
    g := filsg;
    d := filsd
   end;
  end
 end;{construit}
```

```
procedure edite (f: parbre);
 {infixe avec parentheses}
 begin
  if f <> nil then
   with f^ do
   begin
    write('(');
    edite(g);
    write(val);
    edite(d);
    write(')')
   end
 end;{edite}
```

```
procedure postfixe (f: parbre);
 begin
  if f <> nil then
   with f^ do
   begin
    postfixe(g);
    postfixe(d);
    write(val);
   end
 end;{postfixe}
```

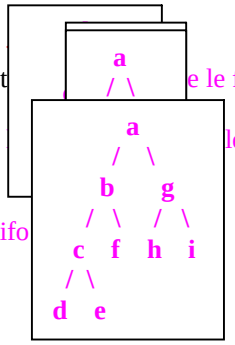
```
procedure prefixe (f: parbre);
 begin
  if f <> nil then
   with f^ do
   begin
    write(val);
    prefixe(g);
    prefixe(d)
   end
 end;{prefixe}
```

```
procedure infixe (f: parbre);
 begin
  if f <> nil then
   with f^ do
   begin
    infixe(g);
    write(val);
    infixe(d);
   end
 end;{infixe}
```

```
procedure Largeur ( f : parbre);
 var Fifo : TList;
 begin
  if f <> nil then
  begin
   Fifo:=TList.Create;  // crée la Fifo
   Fifo.Add( f ); // ajoute la racine f dans Fifo
   while fifo.count<>0 do
   begin
    write(parbre(Fifo.First)^.info); // traitement du
premier
    if parbre(Fifo.First)^.filsG <> nil then
     Fifo.Add(parbre(Fifo.First)^.filsG); // ajoute le fils
```

```
begin {prog-principal}

 arb1:=nil;
 arb2:=nil;
 arb3:=nil;
 arb4:=nil;
 arb5:=nil;
 construit(arb1,nil,nil,'d');
 construit(arb2,nil,nil,'e');
 construit(arb3,arb1,arb2,'c');
 construit(arb4,nil,nil,'f');
 construit(arb5,arb3,arb4,'b');
 {---------------------------------  }
```

```
gauche du premier dans Fifo
   if parbre(Fifo.First)^.filsD
    Fifo.Add(parbre(Fifo.First                    e le fils
droit du premier dans Fifo
   Fifo.delete(0); // supprime                    le
premier)
  end;
  Fifo.Free ; // supprime la Fifo
 end
end;{Largeur}
```




```
arb1:=nil;
arb2:=nil;
arb3:=nil;
construit(arb1,nil,nil,'h');
construit(arb2,nil,nil,'i');
construit(arb3,arb1,arb2,'g');

{--------------------------------}
rac:=nil;
construit(rac,arb5,arb3,'a');
{--------------------------------}

writeln('lecture parenthésée:');
edite(rac);   writeln;
writeln('lecture notation infixée:');
infixe(rac);   writeln;  //dcebfahgj
writeln('lecture notation postfixée:');
postfixe(rac);   writeln; //decfbhjga
writeln('lecture notation préfixée:');
prefixe(rac);   writeln; //abcdefghj
writeln('lecture hiérarchique:');
Largeur(rac);   writeln //abgcfhjde
end.
```

Ci-dessous une unit complète en Delphi de la classe TreeBin :

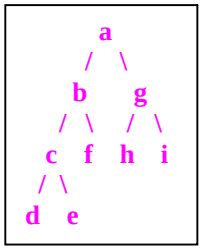
```
unit UTreeBin;

 interface
  uses Classes;
  type
     TreeBin = class
 private
     FInfo : string;
     procedure FreeRecur ( x :TreeBin ) ;
     procedure PreOrdre ( x : TreeBin ; var res :string);
     procedure PostOrdre ( x : TreeBin ; var res :string);
     procedure SymOrdre ( x : TreeBin ; var res :string);
     procedure Hierarchie (x: TreeBin ; var res:string);
 public
      filsG , filsD : TreeBin;
      constructor Create(s:string; fg , fd : TreeBin);
      destructor Liberer;
      function Prefixe : string;
      function Postfixe : string;
      function Infixe : string;
      function Largeur : string;
      property Info : string read FInfo write FInfo;
end;

implementation
{-------------------   Méthodes privé -------------------------}
 procedure TreeBin.FreeRecur ( x : TreeBin ) ;
 // parcours postfixe pour détruire les objets de l'arbre x
 begin
   FreeRecur( x.filsG );
   FreeRecur( x.filsD );
   x.Free
 end;
```

```
procedure TreeBin.PostOrdre (x: TreeBin ;var res:string);
// parcours postfixé d'un arbre x
begin
 if x<>nil then
 begin
   PostOrdre( x.filsG ,res );
   PostOrdre( x.filsD ,res );
   res:=res+x.FInfo
 end
end;

procedure TreeBin.PreOrdre (x: TreeBin ;var res:string);
begin
 if x<>nil then
 begin
  res:=res+x.FInfo;
  PreOrdre( x.filsG ,res );
  PreOrdre( x.filsD ,res );
 end
end;

procedure TreeBin.SymOrdre ( x : TreeBin ;var res:string);
begin
 if x<>nil then
 begin
  SymOrdre ( x.filsG ,res );
  res:=res+x.FInfo;
  SymOrdre ( x.filsD ,res );
 end
end;

procedure TreeBin.Hierarchie (x: TreeBin ;var res:string);
var Fifo : TList;
begin
 if x <> nil then
 begin
  Fifo:=TList.Create;  // crée la Fifo
  Fifo.Add( x ); // ajoute la racine f dans Fifo
  while fifo.count<>0 do
  begin
    res:=res+ TreeBin (Fifo.First).info;
    if TreeBin (Fifo.First).filsG <> nil then
       Fifo.Add(TreeBin (Fifo.First).filsG); // ajoute le fils gauche du premier dans Fifo
    if TreeBin (Fifo.First).filsD <> nil then
      Fifo.Add(TreeBin (Fifo.First).filsD); // ajoute le fils droit du premier dans Fifo
    Fifo.delete(0); // supprime l'élément de rang 0 (le premier)
  end;
  Fifo.Free ; // supprime la Fifo
 end
end;{ Hierarchie }

{-------------------   Méthodes public -------------------------}
constructor TreeBin.Create(s : string ; fg, fd : TreeBin);
begin
 self.Finfo := s;
 self.filsG := fg;
 self.filsD := fd;
end;
```

```
destructor TreeBin.Liberer;
// la destruction de tout l'arbre :
begin
 FreeRecur (self) ;
 self := nil;
end;

function TreeBin.Postfixe:string;
var parcours:string;
begin
 parcours:='';
 PostOrdre( self , parcours );
 result:=parcours
end;

function TreeBin.Prefixe : string;
var parcours:string;
begin
 parcours:='';
 PreOrdre( self , parcours );
 result:=parcours
end;

function TreeBin.Infixe : string;
var parcours:string;
begin
 parcours:='';
 SymOrdre ( self , parcours );
 result:=parcours
end;

function TreeBin.Largeur : string;
var parcours:string;
begin
 parcours:='';
 Hierarchie ( self , parcours );
 result:=parcours
end;

end.
```

Ci-après le programme d'utilisation de la unit sur l'arbre ci-dessous :

```
program PrTreeBin;
 {$APPTYPE CONSOLE}
uses
 SysUtils,
 UTreeBin in 'UTreeBin.pas';
var racine,rac0,rac1,fg,fd : TreeBin;

 begin
  fg:=TreeBin.Create('d',nil,nil);
  fd:=TreeBin.Create('e',nil,nil);
  rac0:=TreeBin.Create('c',fg,fd);
  fd:=TreeBin.Create('f',nil,nil);
  rac1:=TreeBin.Create('b',rac0,fd);
  fg:=TreeBin.Create('h',nil,nil);
  fd:=TreeBin.Create('j',nil,nil);
  rac0:=TreeBin.Create('g',fg,fd);
  racine:=TreeBin.Create('a',rac1,rac0);
```



```
  writeln('lecture notation infixee:');
  writeln(racine.infixe);
  writeln('lecture notation postfixee:');
  writeln(racine.postfixe);
  writeln('lecture notation prefixee:');
  writeln(racine.prefixe);
  writeln('lecture hierarchique:');
  writeln(racine.Largeur);
  readln
 end.
```

Ex-7 : Solution des l'insertion, de la suppression et de la recherche dans une arbre binaire de recherche

Insertion – procédure Placer en Delphi avec les variables dynamiques :

```
procedure placer (var arbre:parbre ; elt:string);
{remplissage récursif de l'arbre binaire de recherche
par adjonctions  successives aux feuilles de l'arbre }
begin
 if arbre = nil then
 begin
  new(arbre);
  arbre^.info:=elt;
  arbre^.filsG :=nil;
  arbre^.filsD :=nil
 end
 else
 if elt <= arbre^.info then placer (arbre^.filsG ,elt)
 else placer (arbre^.filsD , elt);
end;
```

procédure Placer en Delphi avec la classe TreeBinRech héritant de TreeBin (ex-6):

```
interface

type
   TreeBinRech = class ( TreeBin )
      private
         procedure placerArbre (var arbre:TreeBinRech ; elt : string);
      public
         procedure Placer ( elt : string);
    end;

implementation
{--------------------   Méthodes privé -------------------------}
procedure TreeBinRech.placerArbre (var arbre:TreeBinRech ; elt:string);
{remplissage récursif de l'arbre binaire de recherche
par adjonctions  successives aux feuilles de l'arbre }
begin
 if not Assigned(arbre) then
    arbre:=TreeBinRech.CreerTreeBin( elt )
 else
 if elt <= arbre.info then placerArbre (TreeBinRech(arbre.filsG) ,elt)
  else placerArbre (TreeBinRech(arbre.filsD) , elt);
end;

{--------------------   Méthodes public -------------------------}
```

```
 procedure TreeBinRech.Placer ( elt : string);
 begin
    placerArbre ( self, elt )
 end;

end.
```

Procédure Chercher en Delphi avec les variables dynamiques :

```
function Chercher( arbre:parbre; elt:string) : parbre;
begin
 if arbre = nil then
 begin
     result := nil;
     writeln('élément non trouvé')
 end
 else
   if elt = arbre^.info then result := arbre //l'élément est dans ce noeud
   else
     if elt < arbre^.info then
        result := Chercher(arbre^.filsG , elt) //on cherche à gauche
     else
        result := Chercher(arbre^.filsD , elt) //on cherche à droite
end;
```

procédure Chercher en Delphi avec la classe TreeBinRech :

```
interface

type
   TreeBinRech = class ( TreeBin )
      private
         procedure placerArbre (var arbre:TreeBinRech ; elt : string);
         function ChercherArbre( arbre:TreeBinRech; elt:string) : TreeBinRech;
      public
         procedure Placer ( elt : string);
         procedure Chercher ( elt : string);
    end;

implementation
{-------------------   Méthodes privé ------------------------}
function TreeBinRech.ChercherArbre( arbre:TreeBinRech; elt:string) : TreeBinRech;
begin
 if not Assigned(arbre) then
 begin
     result := nil;
     writeln('élément non trouvé')
 end
 else
   if elt = arbre.info then result := arbre //l'élément est dans ce noeud
   else
     if elt < arbre.info then
        result := ChercherArbre(TreeBinRech(arbre.filsG) , elt) //on cherche à gauche
     else
        result := ChercherArbre(TreeBinRech(arbre.filsD) , elt) //on cherche à droite
end;

{-------------------   Méthodes public ------------------------}
```

```
procedure TreeBinRech.Chercher ( elt : string);
begin
   ChercherArbre( self , elt )
end;

end.
```

procédure Supprimer avec les variables dynamiques :

```
function PlusGrand ( arbre  : parbre ) : parbre;
begin
 if arbre^.filsD = nil then
     result := arbre
 else
   result := PlusGrand( arbre^.filsD ) //on descend à droite
end;

procedure Supprimer (var arbre:parbre; elt:string ) ;
var Node,Loc:parbre;
begin
  if arbre <> nil then
    if elt < arbre^.info  then
       Supprimer (arbre^.filsG, elt )
   else
    if elt > arbre^.info  then
       Supprimer (arbre^.filsD, elt )
    else // elt = arbre^.info
    if arbre^.filsG = nil  then
    begin
       Loc:=arbre;
       arbre := arbre^.filsD;
       dispose(Loc)
    end
    else
       if arbre^.filsD = nil then
       begin
         Loc:=arbre;
         arbre := arbre^.filsG;
         dispose(Loc)
      end
      else
      begin
        Node := PlusGrand ( arbre^.filsG );
        Loc:=Node;
        arbre^.info := Node^.info;
        arbre^.filsG :=Node^.filsG;
        dispose(Loc)
      end
end;
```

procédure Supprimer en Delphi avec la classe TreeBinRech :

```
interface
type
   TreeBinRech = class ( TreeBin )
      private
         procedure placerArbre (var arbre:TreeBinRech ; elt : string);
         function ChercherArbre( arbre:TreeBinRech; elt:string) : TreeBinRech;
         function SupprimerElt( arbre:TreeBinRech; elt:string) : TreeBinRech;
```

```
        function PlusGrand(arbre: TreeBinRech): TreeBinRech;
      public
        procedure Placer ( elt : string);
        procedure Chercher ( elt : string);
        procedure Supprimer ( elt : string);
    end;
implementation
{-------------------   Méthodes privé -------------------------}
function TreeBinRech.PlusGrand(arbre: TreeBinRech): TreeBinRech;
 begin
  if not Assigned(Arbre.filsD) then
   result:=Arbre //c'est le pus grand élément
  else
   result:=PlusGrand (TreeBinRech(Arbre.filsD))
 end;

function TreeBinRech.SupprimerElt ( arbre:TreeBinRech; elt:string) : TreeBinRech;
var Noeud:TreeBinRech;
 begin
  if  not Assigned(Arbre) then
   writeln('element ',Elt,' non trouve dans l''arbre')
  else
   if Elt  < Arbre.Info  then
    SupprimerElt (TreeBinRech(Arbre.filsG), Elt ) //on cherche à gauche
   else
    if Elt  >  Arbre.Info  then
     SupprimerElt ( TreeBinRech(Arbre.filsD), Elt ) //on cherche à droite
    else //l'élément est dans ce noeud
    begin
     if Arbre.filsG = nil then //sous-arbre gauche vide
     begin
      Noeud :=Arbre;
      Arbre := TreeBinRech(Arbre.filsD); //remplacer arbre par son sous-arbre droit
     end
     else
      if Arbre.filsD = nil then //sous-arbre droit vide
      begin
       Noeud :=Arbre;
       Arbre :=TreeBinRech(Arbre.filsG); //remplacer arbre par son sous-arbre gauche
      end
      else //le noeud a deux descendants
      begin
       Noeud := PlusGrand( TreeBinRech(Arbre.filsG )); //Node = le max du fils gauche
       Arbre.Info:= Noeud.Info; //remplacer etiquette
       Arbre.filsG := Noeud.filsG;
      end ;
     Noeud.Free; //on supprime ce noeudil;
     writeln('element ',Elt,' supprime !')
    end
   end;
  {-------------------   Méthodes public -------------------------}
   procedure TreeBinRech.Supprimer ( elt : string);
   begin
     SupprimerElt ( self , elt )
   end;

 end.
```

Programme Delphi complet

Ci-dessous un programme complet en Delphi (avec variables dynamiques) à exécuter tel quel avec Delphi en mode console; il est conseillé au lecteur de ré-écrire ce programme en utilisant la classe TreeBinRech décrite ci-haut :

```
program arbre_binaire_Rech;
{remplissage d'un arbre binaire de recherche
aussi dénommé arbre binaire ordonné horizontalement  }

{$APPTYPE CONSOLE}

uses   sysutils;
type
   pointeur = ^noeud;
   noeud = record
                 info:string;
                 filsGauche:pointeur;
                 filsDroit:pointeur
             end;
var
   racine,tree:pointeur;

procedure placer_arbre(var arbre:pointeur; elt:string);
{remplissage récursif de l'arbre binaire de recherche}
begin
 if arbre=nil then
 begin
  new(arbre);
  arbre^.info:=elt;
  arbre^.filsGauche:=nil;
  arbre^.filsDroit:=nil
 end
 else
 { - tous les éléments "info" de tous les noeuds du sous-arbre de gauche
     sont inférieurs ou égaux à l'élément "info" du noeud en cours (arbre)

   - tous les éléments "info" de tous les noeuds du sous-arbre de droite
     sont supérieurs à l'élément "info" du noeud en cours (arbre)
 }
 if elt<=arbre^.info then placer_arbre(arbre^.filsGauche,elt)
 else placer_arbre(arbre^.filsDroit,elt);
end;


procedure infixe(branche:pointeur);
{lecture symétrique de l'arbre binaire}
begin
  if branche<>nil then
  begin
   infixe(branche^.filsGauche);
   write(branche^.info,' ');
   infixe(branche^.filsDroit);
  end
end;

function ChercherArbre( arbre:pointeur; elt:string):pointeur;
begin
```

```
 if arbre=nil then
 begin
   ChercherArbre:=nil;
   writeln('élément: '+elt+' non trouvé.')
 end
 else
  if elt = arbre^.info then ChercherArbre:=arbre
  else
   if elt < arbre^.info then ChercherArbre:=ChercherArbre(arbre^.filsGauche,elt)
   else ChercherArbre:=ChercherArbre(arbre^.filsDroit,elt)
end;

function PlusGrand ( arbre  : pointeur ) : pointeur;
// renvoie le plus grand élément de l'arbre
begin
 if arbre^.filsDroit = nil then  result := arbre
 else  result := PlusGrand ( arbre^.filsDroit ) //on descend à droite
end;

procedure Supprimer (var arbre:pointeur; elt:string ) ;
var Node,Loc:pointeur;
begin
  if arbre <> nil then
    if elt < arbre^.info  then
       Supprimer (arbre^.filsGauche, elt )
   else
    if elt > arbre^.info  then
       Supprimer (arbre^.filsDroit, elt )
    else // elt = arbre^.info
    if arbre^.filsGauche = nil  then
    begin
       Loc:=arbre;
       arbre := arbre^.filsDroit;
       dispose(Loc)
    end
    else
       if arbre^.filsDroit = nil then
       begin
         Loc:=arbre;
         arbre := arbre^.filsGauche;
         dispose(Loc)
      end
      else
      begin
        Node := PlusGrand ( arbre^.filsGauche );
        Loc:=Node;
        arbre^.info := Node^.info;
        arbre^.filsGauche :=Node^.filsGauche;
        dispose(Loc)
      end
end;

begin
  racine:=nil;
  {soit la liste entrée : e d f a c b u w }
  placer_arbre(racine,'e');
  placer_arbre(racine,'d');
  placer_arbre(racine,'f');
  placer_arbre(racine,'a');
  placer_arbre(racine,'c');
  placer_arbre(racine,'b');
  placer_arbre(racine,'u');
```

```
  placer_arbre(racine,'w');
  {on peut aussi entrer 8 éléments au clavier
  for i:=1 to 8 do
  begin
    readln(entree);
    placer_arbre(racine,entree);
  end; }
  supprimer(racine, 'b');
  writeln('parcours infixé (ou symétrique):');
  infixe(racine);
  writeln;
  tree:=ChercherArbre(racine,'c');
  if tree<>nil then writeln( 'recherche de "c" : ok');
  tree:=ChercherArbre(racine,'g');
  if tree<>nil then writeln( 'recherche de "g" : ok');
  { Notons que le parcours infixé produit une liste des
    éléments info, classée par ordre croissant
  }
end.
```

Chapitre 5 : Programmation objet et événementielle

5.1 Introduction à la programmation orientée objet (POO)

- concepts fondamentaux de la POO
- initiation à la conception orientée objet (OOD,UML...)

5.2 Programmez objet avec Delphi

- Description générale de Delphi
- Modules dans Delphi
- Delphi et la POO
- Les propriétés en Delphi

5.3 Polymorphisme avec Delphi

- Polymorphisme d'objet
- Polymorphisme de méthodes
- Polymorphisme de classe abstraite
- Exercice traité sur le polymorphisme

5.4.Programmation événementielle et visuelle

- programmation visuelle basée sur les pictogrammes
- programmation orientée événements
- normalisation du graphe événementiel
- tableau des actions événementielles
- interfaces liées à un graphe événementiel
- avantage et modèle de développement RAD visuel
- Notice sur les interfaces de communication

5.5.les événements avec Delphi

- Pointeur de méthode
- Affecter un pointeur de méthode
- Un événement est un pointeur de méthode
- Quel est le code engendré
- Exercice-récapitulatif
- Notice méthodologique pour créer un nouvel événement
- Code pratique : une pile lifo événementielle

- Exemple traité : Un éditeur de texte

5.6.programmation défensive : les exceptions

- **Notions de défense et de protection**

  Outils participant à la programmation défensive
  Rôle et mode d'action d'une exception
  Gestion de la protection du code
  - Fonctionnement sans incident
  - Fonctionnement avec incident
  - Effets dus à la position du bloc except...end
  - Fonctionnement sans incident
  - Fonctionnement avec incident
  - Interception d'une exception d'une classe donnée
  - Ordre dans l'interception d'une exception
  - Interception dans l'ordre de la hiérarchie
  - Interception dans l'ordre inverse

- **Traitement d'un exemple de protections**

  Le code de départ de l'unité
  Code de la version.1 (premier niveau de sécurité)
  Code de la version.2 (deuxième niveau de sécurité)

❖ Code pratique : une pile Lifo avec exception

5.1 Introduction à la programmation orientée objet

Plan du chapitre:

Introduction

1. Concepts fondamentaux de La P.O.O

1.1 les objets
1.2 les classes
1.3 L'héritage

2. Introduction à la conception orientée objet:

2.1 La méthode de conception OOD
2.2 Notation UML de classes et d'objets
SCHEMA UML DE CLASSE
VISIBILITE DES ATTRIBUTS ET DES METHODES
SCHEMA UML D'UN OBJET
SCHEMA UML DE L'HERITAGE
SCHEMA UML DES ASSOCIATIONS
UNE ASSOCIATION PARTICULIERE : L'AGREGATION
NOTATION UML DES MESSAGES

2.3 Attitudes et outils méthodologiques

polymorphisme d'objet

polymorphisme par héritage de méthode

polymorphisme par héritage de classes abstraites

polymorphisme par implémentation d'interfaces

Introduction

Le lecteur qui connaît les fondements de ce chapitre peut l'ignorer et passer au chapitre suivant. Dans le cas contraire, **la programmation visuelle** étant intimement liée aux outils et aux concepts **objets**, ce chapitre est un minimum incontournable.

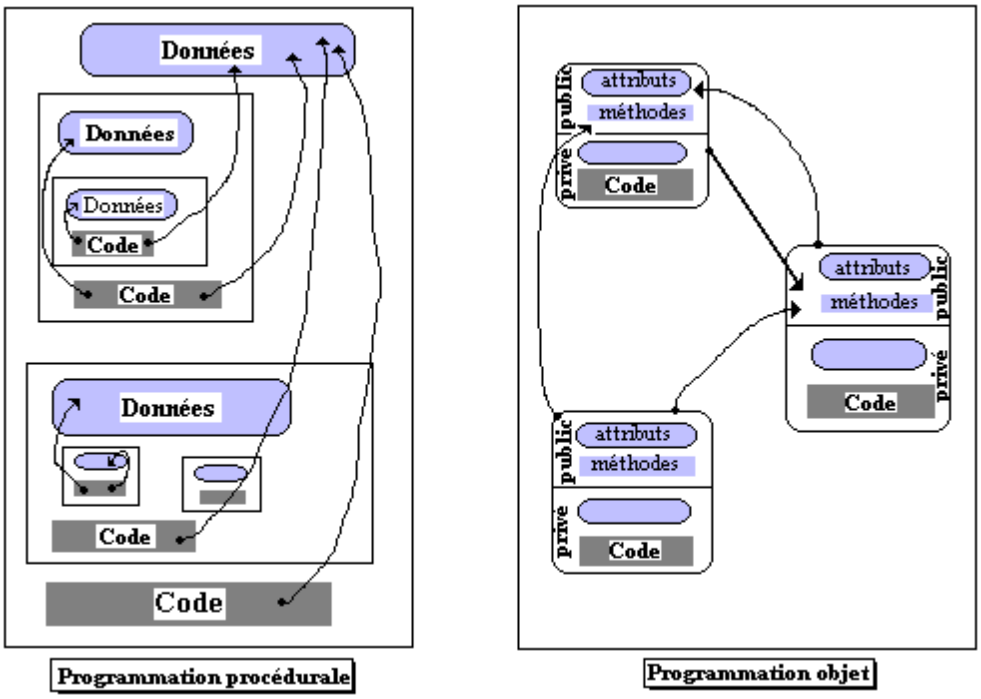
La programmation classique ou *procédurale* telle que le débutant peut la connaître à travers des langages de programmation comme Pascal, C etc... traite les programmes comme un ensemble de données sur lesquelles agissent des procédures. Les procédures sont les éléments actifs et importants, les données devenant des éléments passifs qui traversent *l'arborescence de programmation* procédurale en tant que flot d'information.

Cette manière de concevoir les programmes reste proche des machines de Von Neuman et consiste en dernier ressort à traiter indépendamment les données et les algorithmes (traduits par des procédures) sans tenir compte des relations qui les lient.

En introduisant la notion de modularité dans la programmation structurée descendante, l'approche diffère légèrement de l'approche habituelle de la programmation algorithmique classique. Nous avons défini des *machines abstraites* qui ont une autonomie relative et qui possèdent leurs propres structures de données; la conception d'un programme relevait dès lors essentiellement de la description des interactions que ces machines ont entre elles.

La programmation orientée objet relève d'une conception ascendante définie comme des "**messages**" échangés par des entité de base appelées objets.

comparaison des deux topologies de programmation

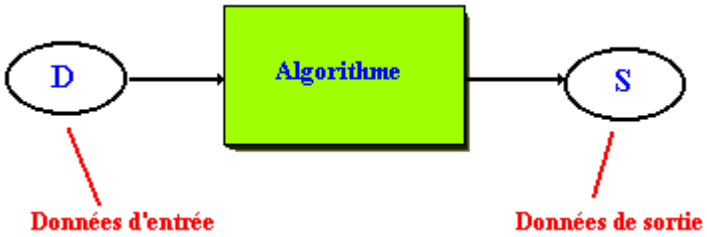


Les langages objets sont fondés sur la connaissance d'une seule catégorie d'entité informatique : l'objet. Dans un objet, traditionnellement ce sont les données qui deviennent

prépondérantes. On se pose d'abord la question : "de quoi parle-t-on ?" et non pas la question "que veut-on faire ?", comme en programmation algorithmique. C'est en ce sens que les machines abstraites de la programmation structurée modulaire peuvent être considérées comme des pré-objets.

En fait la notion de TAD est utilisée dans cet ouvrage comme spécification d'un objet, en ce sens nous nous préoccupons essentiellement des services offerts par un objet indépendamment de sa structure interne.

Programmation structurée :



1. Concepts fondamentaux de La P.O.O

Nous écrirons P.O.O pour : programmation orientée objet.

Voici trois concepts qui contribuent à la puissance de la P.O.O.

- Concept de **modélisation** à travers la notion de classe et **d'instanciation** de ces classes.
- Concept **d'action** à travers la notion d'envoi de messages et de méthodes à l'intérieur des objets.
- Concept de construction par **réutilisation et amélioration** par l'utilisation de la notion d'héritage.

1.1 les objets

Objet

Un **module** représente ***un objet ou une classe d'objet*** de l'espace du problème et non une étape principale du processus total, comme en programmation descendante.

Recenser les objets du monde réel

Lors de l'analyse du problème, on doit faire l'état de l'existant en recensant les objets du monde réel. On établit des classes d'objets et pour chaque objet on inventorie les connaissances que l'on a sur lui :

- Les connaissances déclaratives,
- les connaissances fonctionnelles,
- l'objet réel et les connaissances que l'on a sur lui sont regroupés dans une même entité.

On décrit alors les systèmes en classes d'objets plutôt qu'en terme de fonctions.

Par Exemple : **Une application de gestion bancaire est organisée sur les objets comptes, écritures, états.**

Les objets rassemblent une partie de la connaissance totale portant sur le problème. Cette connaissance est répartie sur tous les objets sous forme déclarative ou procédurale.

Les objets sont décrits selon le modèle des structures abstraites de données (TAD) : ils constituent des boîtes noires dissimulant leur implantation avec une interface publique pour les autres objets. Les interactions s'établissant à travers cette interface.

Vocabulaire objet

Encapsulation

c'est le fait de réunir à l'intérieur d'une même entité (*objet*) le code (*méthodes*) + données (*champs*).

Il est donc possible de masquer les informations d'un objet aux autres objets.

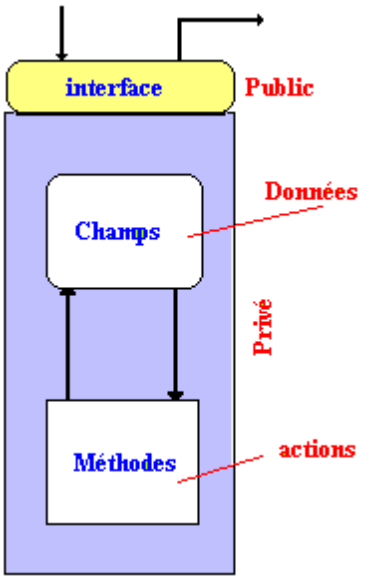
Deux niveaux d'encapsulation sont définis :

Privé

les **champs** et les **méthodes masqués** sont dans la partie privée de l'objet.

Public

les **champs** et les **méthodes visibles** sont dans la partie interface de l'objet.



Les notions de **privé** et de **public** comme dans un objet n'ont trait qu'à la communication entre deux objets, à l'intérieur d'un objet elles n'ont pas cours.

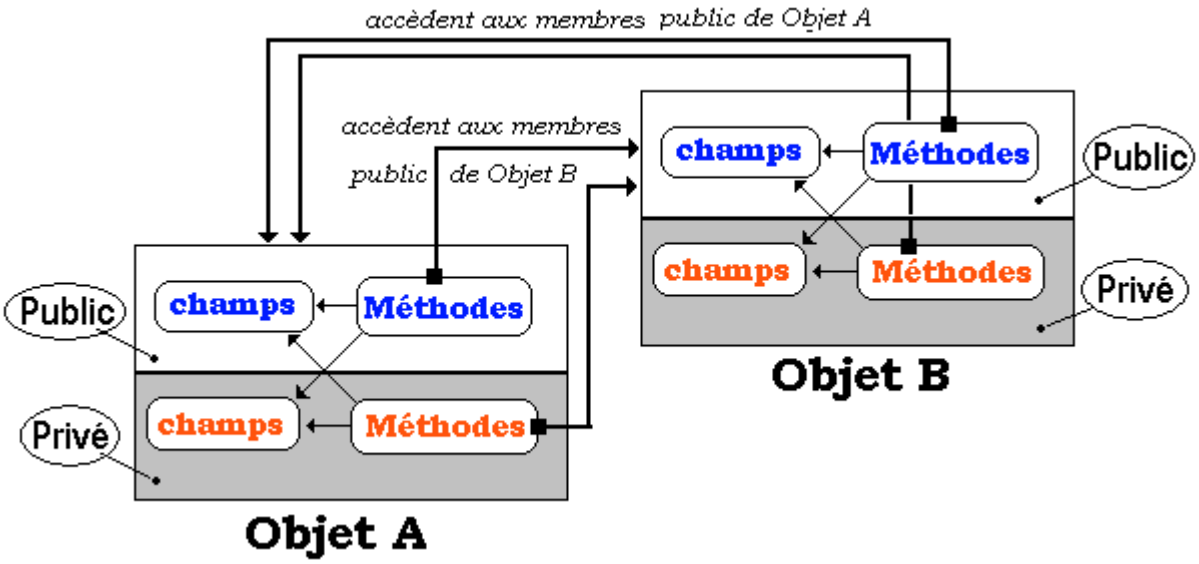


Figure sur la visibilité entre deux objets
Les méthodes de public ou privé de l'objet A accèdent et peuvent utiliser les méthodes et les champs public de B.
Les méthodes de public ou privé de l'objet B accèdent et peuvent utiliser les méthodes et les champs public de A.

1.2 les classes

Postulons une analogie entre les objets matériels de la vie courante et les objets informatiques. Un objet de tous les jours est souvent obtenu à partir d'un moule industriel servant de modèle pour en fabriquer des milliers. Il en est de même pour les objets informatiques.

Classe

Une **classe** est une sorte de ***moule ou de matrice*** à partir duquel sont engendrés les objets réels qui s'appellent des ***instances*** de la classe considérée.

Une classe contient

Des attributs (ou champs, ou variables d'instances).
Les attributs de la classe décrivent la structure de ses instances (les objets).

Des méthodes (ou opérations de la classe).
Les méthodes décrivent les opérations qui sont applicables aux instances de la classe.

Membres

les **attributs** et les **méthodes** d'une classe sont des **membres** de la classe.

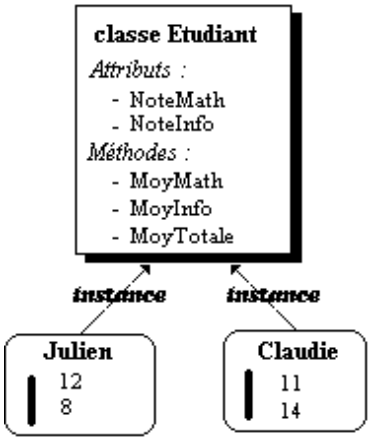
Instance

Un objet de classe A est appelé aussi une **instance** de classe A, l'opération de construction d'un objet s'appelle alors **l'instanciation**.

Remarque

En POO, programmer revient donc à décrire des classes d'objets, à caractériser leur structure et leur comportement, puis à instancier ces classes pour créer des objets réels. Un objet réel est matérialisé dans l'ordinateur par une zone de mémoire que les données et son code occupent.

Un exemple : des étudiants
Supposons que chaque étudiant soit caractérisé par sa note en mathématiques (NoteMath) et sa note en informatique (NoteInfo). Un étudiant doit pouvoir effectuer éventuellement des opérations de calcul de ses moyennes dans ces deux matières (MoyMath, MoyInfo)et connaître sa moyenne générale calculée à partir de ces deux notes (MoyTotale).



La classe Etudiant a été créée. Elle ne possède que les **attributs** *NoteMath* et *NoteInfo*. Les **méthodes** de cette classe sont par exemple *MoyMath*, *MoyInfo*, *MoyTotale*.

Nous avons créé deux **objets** étudiants de la classe Etudiant (deux **instances** : Julien et Claudie).

1.3 L'héritage

Dans un **LOO** (**L**angage **O**rienté **O**bjet), il existe une particularité dans la façon d'organiser ses classes : l'héritage de propriétés. L'objectif est de construire de nouvelles classes en **réutilisant** des attributs et des méthodes de classes déjà existantes.

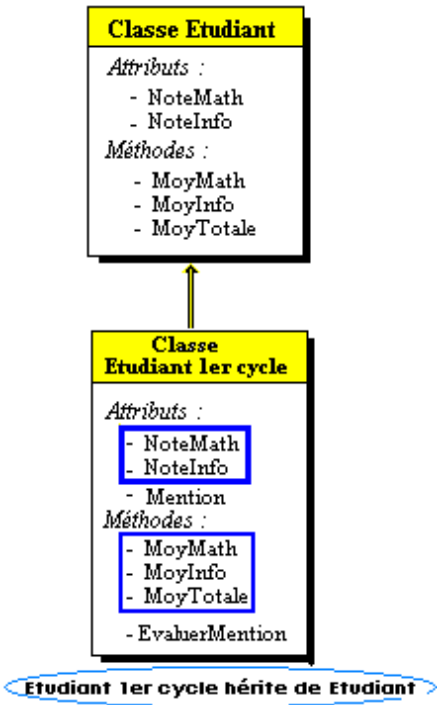
Héritage

C'est un mécanisme très puissant qui permet de décrire des structures génériques en transmettant depuis l'intérieur d'une même classe toutes les propriétés communes à toutes les " sous-classes " de cette classe. Par construction toutes les sous-classes d'une même classe possèdent toutes les attributs et les méthodes de leur classe parent.

Propriétés de l'héritage

- Les attributs et les méthodes peuvent être modifiés au niveau de la sous-classe qui hérite.
- Il peut y avoir des attributs et/ou des méthodes supplémentaires dans une sous-classe.
- Une classe A qui hérite d'une classe B dispose implicitement de tous les attributs et de toutes les méthodes définis dans la classe B.
- Les attributs et les méthodes définis dans A sont prioritaires par rapport aux attributs et aux méthodes de même nom définis dans

Exemple :
La classe " *Etudiant premier cycle*" héritant de la classe *Etudiant* précédemment définie.



Tout se passe comme si toute la classe *Etudiant* était recopiée dans la sous-classe *Etudiant premier cycle* (même si l'implémentation n'est pas faite ainsi).

La nouvelle classe dispose d'un attribut supplémentaire (Mention) et d'une méthode supplémentaire (EvaluerMention).

```
type Tmention=(Passable, Abien, Bien, Tbien);
Etudiant = class
    NoteMath : real;
    NoteInfo : real;
    procedure MoyMath(UneNote:real);
    procedure MoyInfo(UneNote:real);
    function MoyTotale : real ;
end;
 Etudiant1erCycle = class(Etudiant)
    Mention: Tmention ;
    function EvaluerMention: Tmention;
end;
```

Ceci est une implantation possible de la signature de la classe en :

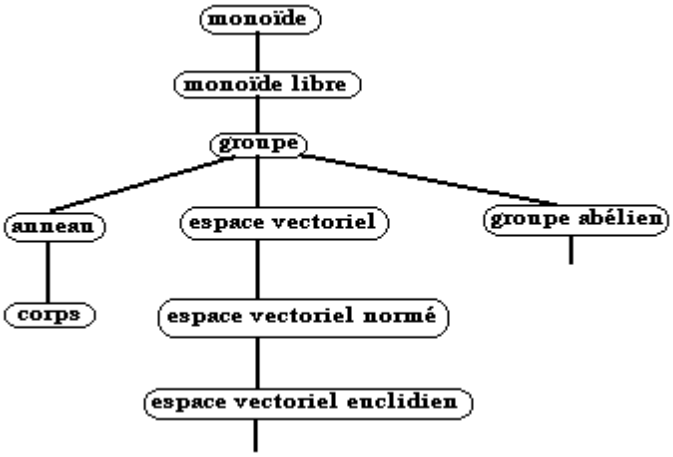
Delphi

```
class Etudiant {
   public float NoteMath;
   public float NoteInfo;
   public void MoyMath(float UneNote){
   ...     }
   public void MoyInfo(float UneNote){
   ...     }
   public float MoyTotale(){
   ...     }
 }// fin classe Etudiant
public class Etudiant1erCycle extends Etudiant{
   public String Mention;
   public String EvaluerMention( ){
   ...     }
   public static void Main(String[ ] args){
   ...     }
}// fin
```

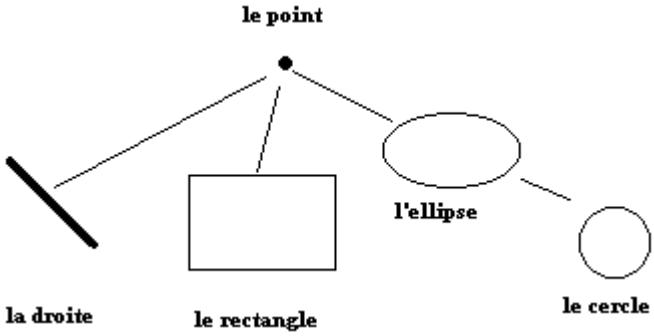
Ceci est une implantation possible de la signature de la classe en :

Exemples d'héritage dans d'autres domaines :

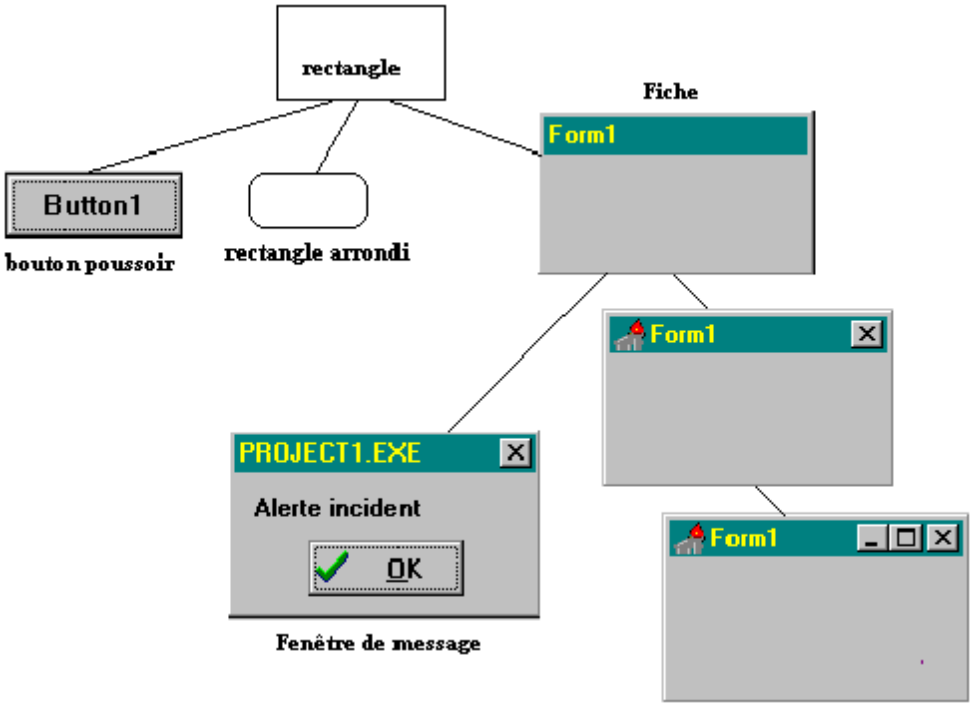
Héritage dans les structures mathématiques :



Héritage de figures de base en géométrie affine :



Héritage d'objets graphiques dans un système multi-fenêtré fictif :



2. Introduction à la conception orientée objet

L'attitude est ici résolument sous-tendue par un double souci : fournir des outils méthodologiques rationalisant l'effort de production du logiciel, sans que leur lourdeur rebute l'étudiant non professionnel et masque ainsi l'intérêt de leur utilisation. L'expérience d'enseignement de l'auteur avec des débutants a montré que si les étudiants sont appelés à développer sans outils méthodiques, ils pratiquent ce qu'appelle J.Arsac " la grande bidouille ". Mais dans le cas contraire, l'apprentissage détaillé de trop de méthodes strictes bien qu'efficaces (OOD, OMT, HOOD, UML,...)finit par ennuyer l'étudiant ou du moins par endormir son sens de l'intérêt. Dans ce dernier cas l'on retrouve " la grande bidouille " comme étape finale. Le chemin est donc étroit et il appartient à chaque enseignant de doser en fonction de l'auditoire l'utile et le superflu.

Nous utilisons ici un de ces dosages pour montrer à l'étudiant comment écrire des programmes avec des objets sans être un grand spécialiste. Une aide irremplaçable à cet égard nous sera fournie par l'environnement de développement visuel Delphi.

2.1 La méthode de conception OOD simplifiée

La méthode O.O.D (object oriented design) de G.Booch propose *5 étapes* dans l'établissement d'une conception orientée objet. Ces étapes n'ont pas obligatoirement à être enchaînées dans l'ordre dans lequel nous les citons dans le paragraphe suivant. C'est cette souplesse qui nous a fait choisir la démarche de G.Booch, car cette méthode est fondamentalement incrémentale et n'impose **pas un cadre trop précis et trop rigide** dans son application. Cette démarche se révèle être utile pour un débutant et lui permettra de fabriquer en particulier des prototypes avec efficacité sans trop surcharger sa mémoire.

Identifier les objets et leurs attributs

On cherchera à identifier les objets du monde réel que l'on voudra réaliser.

Conseil : *Il faut identifier les propriétés caractéristiques de l'objet (par l'expérience, l'intuition,...).On pourra s'aider d'une description textuelle (en langage naturel) du problème. La lecture de cette prose aidera à déduire les bons candidats pour les noms à utiliser dans cette description ainsi que les propriétés des adjectifs et des autres qualifiants.*

Identifier les opération

On cherchera ensuite à identifier les actions que l'objet subit de la part de son environnement et qu'il provoque sur son environnement.

Conseil : *Les verbes utilisés dans la description informelle (textuelle) fournissent de bons indices pour l'identification des opérations. Si nécessaire, c'est à cette étape que l'on pourra définir les conditions d'ordonnancement temporel des opérations (les événements ayant lieu).*

Etablir la visibilité

L'objet étant maintenant identifié par ses caractéristiques et ses opérations, on définira ses relations avec les autres objets.

Conseil : *On établira quels objets le voient et quels objets sont vus par lui (les spécialistes disent alors qu'on insère l'objet dans la topologie du projet). Il faudra prendre bien soin de définir ce qui est visible ou non, quitte à y revenir plus tard si un choix s'est révélé ne pas être judicieux.*

Etablir l'interface

Dès que la visibilité est acquise, on définit l'interface précise de l'objet avec le monde extérieur.

Conseil : *Cette interface définit exactement quelles fonctionnalités sont accessibles et sous quelles formes. Cette étape doit pouvoir être décrite sous notation formelle; les TAD sont l'outil que nous utiliserons à cette étape de conception.*

Implémenter les objets

La dernière étape consiste à implanter les objets en écrivant le code.

Conseil : *Cette étape peut donner lieu à la création de nouvelles classes correspondant par exemple à des nécessités d'implantation. Le code en général correspond aux spécifications concrètes effectuées avec les TAD, ou à la traduction des algorithmes développés par la méthode structurée. Lors de cette étape, on identifiera éventuellement de nouveaux objets de plus bas niveau d'abstraction qui ne pouvaient pas être analysés en première lecture(ceci provoquant l'itération de la méthode à ces niveaux plus bas).*

Nous n'opposons pas cette méthode de conception à la méthode structurée modulaire. Nous la considérons plutôt comme complémentaire (en appliquant à des débutants une idée contenue dans la méthode HOOD). La méthode structurée modulaire sert à élaborer des algorithmes classiques comme des actions sur des données. La COO permet de définir le monde de l'environnement de façon modulaire. Nous réutiliserons les algorithmes construits dans des objets afin de montrer la complémentarité des deux visions.

2.2 Notation UML de classes et d'objets

La notion d'un langage de modélisation standard pour tout ce qui concerne les développements objets a vu le jour en 1997 il s'agit d'UML (Unified Modeling Language).

UML n'est pas une méthode, mais une notation graphique et un métamodèle.

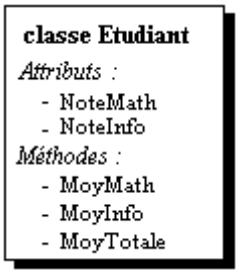
Nous allons fournir ici les principaux schémas d'UML permettant de décrire des démarches de conception objets simples. Le document de spécification de la version 1.4 d'UML par l'OMG (l'Object Management Group) représente 1400 pages de définitions sémantiques et de notations; il n'est donc pas question ici de développer l'ensemble de la notation UML (que d'ailleurs l'auteur ne possède pas lui-même).

Nous nous attacherons à détailler les diagrammes de base qui pourront être utilisés par la suite dans le reste du document.

Nous nous limiterons aux notations relatives aux classes, aux objets, et à l'héritage.

SCHEMA UML DE CLASSE

Notations UML possibles d'une classe :
Reprenons l'exemple précédent avec la classe étudiant :



trois notations UML possibles

Simplifiée	*Avec attributs*	*Attributs et méthodes*
Etudiant	**Etudiant** - NoteMath - NoteInfo	**Etudiant** - NoteMath - NoteInfo + MoyMath + MoyInfo + MoyTotale

Deux autres notations UML plus complète pour la même classe

Attributs et méthodes typées	*Attributs typés et méthodes typées*
Etudiant - NoteMath - NoteInfo + MoyMath () : **réel** + MoyInfo () : **réel** + MoyTotale () : **réel**	**Etudiant** - NoteMath : **réel** - NoteInfo : **réel** + MoyMath () : **réel** + MoyInfo () : **réel** + MoyTotale () : **réel**

Une classe abstraite est notée :

Etudiant
{ abstraite }

VISIBILITE DES ATTRIBUTS ET DES METHODES

Notation préfixée UML pour trois niveaux de visibilité (+, - , # , $) :

Pour les attributs :

public	*privé*	*protégé*
+ Attribut1 : DeType1	- Attribut2 : DeType2	# Attribut3 : DeType3

Pour les méthodes :

public	*privé*	*protégé*
+ Methode1 ( ): DeType1	- Methode2 ( ) : DeType2	# Methode3 ( ) : DeType3

Méthode de classe
$ Methode4 ( ) : DeType4

Explicitation dans la classe Etudiant :

Etudiant
- NoteMath : **réel**
- NoteInfo : **réel**
+ MoyMath () : **réel**
+ MoyInfo () : **réel**
+ MoyTotale () : **réel**

- Dans la classe **étudiant** les deux attributs **NoteMath** et **NoteInfo** sont de type réel et sont **privés** (préfixe -).
- Les trois méthodes de calcul de moyennes sont **publiques** (préfixe +).

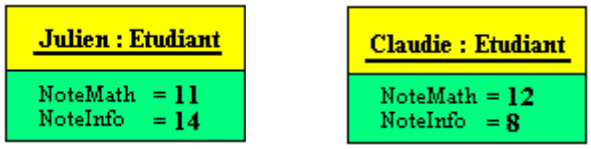
SCHEMA UML D'UN OBJET

Notations UML pour deux objets étudiants instanciés à partir de la classe Etudiant :

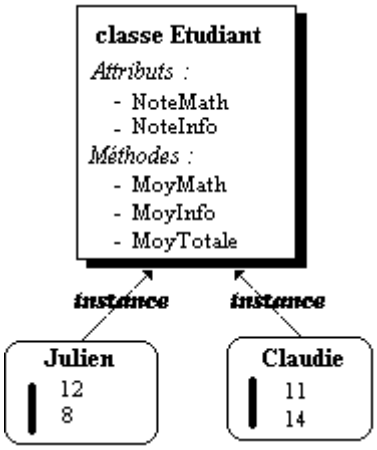
Schéma UML simplifié :

Julien : Etudiant **Claudie : Etudiant**

Schéma UML avec valeur des attributs :

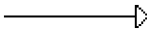


Ces notations correspondent à l'exemple ci-dessous :

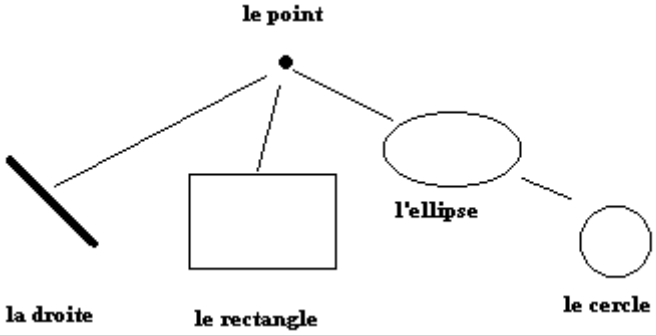


SCHEMA UML DE L'HERITAGE

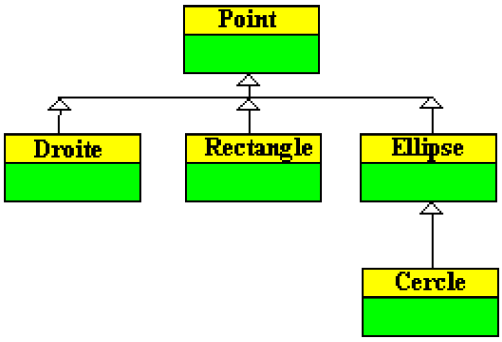
Notation UML de l'héritage :



Soit pour l'exemple de hiérarchie de classes fictives ci-dessous :



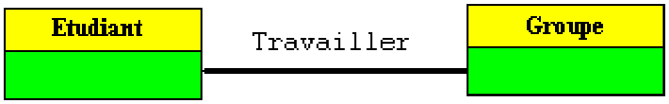
La notation UML suivante :



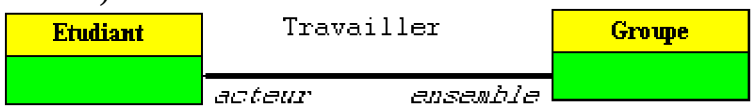
SCHEMA UML DES ASSOCIATIONS

Une association binaire (ou plus généralement n-aire), représente un lien conceptuel entre deux classes. Par exemple un étudiant travaille dans un groupe (association entre la classe **Etudiant** et la classe **Groupe**).

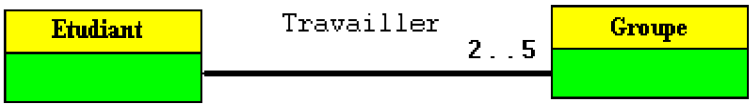
Une association peut être dénotée par une expression appelée nom d'association (nommé **Travailler** ci-dessous) :



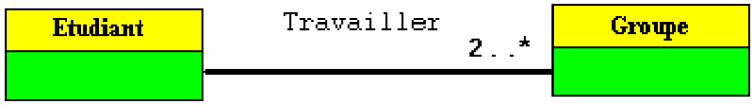
Chaque association possède donc deux extrémités appelées aussi rôles, il est possible de nommer les extrémités (nom de rôles, ci-dessous un étudiant est un acteur travaillant dans un groupe qui est un ensemble) :



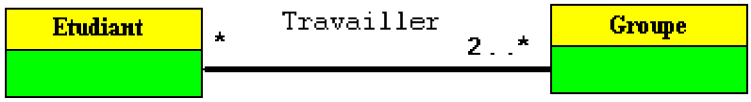
Une association peut posséder une multiplicité qui représente sous forme d'un intervalle de nombres entiers a..b, le nombre d'objets de la classe d'arrivée qui peut être mis en association avec un objet de la classe de départ. Supposons qu'un étudiant doive s'inscrire à au moins 2 groupes de travail et au plus à 5 groupes, nous aurons le schéma UML suivant :



La présence d'une étoile dans la multiplicité indiquant un nombre quelquonque (par exemple un étudiant peut s'inscrire à au moins 2 groupes sans limite supérieure):



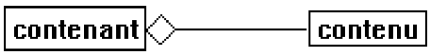
Par exemple pour dénoter en UML le fait qu'un nombre quelconque d'étudiants doit travailler dans au moins deux groupes nous écrirons:



UNE ASSOCIATION PARTICULIERE : L'AGREGATION

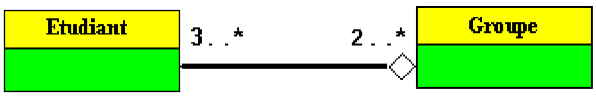
Une agrégation est une association correspondant à une relation qui lorsqu'elle est lue dans un sens signifie "est une partie de" et lorsqu'elle est lue dans l'autre sens elle signifie "est composé de". UML disposant de plusieurs raffinements possibles nous utiliserons l'agrégation comme composition par valeur en ce sens que la construction du tout implique la construction automatique de toutes les parties et que la destruction du tout entraîne en cascade la destruction de chacune de ses parties.

Notation UML de l'agrégation



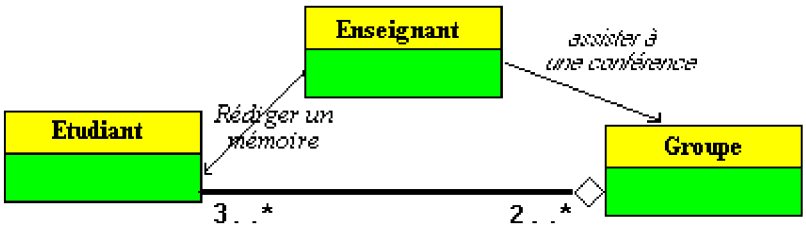
Exemple :

un groupe contient au moins 3 étudiants et chaque étudiant doit s'inscrire à au moins 2 groupes :



NOTATION UML DES MESSAGES

Un message envoyé par une classe à une autre classe est représenté par une flèche vers la classe à qui s'adresse le message, le nommage de la flèche indique le message à exécuter :



2.3 Attitudes et outils méthodologiques

Afin d'utiliser une méthodologie pratique et rationnelle, nous énumérons au lecteur les outils que nous utilisons selon les besoins, dans le processus d'écriture d'un logiciel.

En tout premier la notion de module : C'est la décomposition d'un logiciel en sous-ensembles que l'on peut changer comme des pièces d'un patchwork.
La notion de cycle de vie du logiciel : Développer un logiciel ce n'est pas seulement écrire du Pascal, de l'Ada etc...
Utiliser des TAD : Un type abstrait de données correspond très exactement à l'interface d'un module. Il renforce la méthodologie modulaire.
La programmation structurée par machines abstraites : On se sert d'une méthode de conception descendante et modulaire des algorithmes utiles pour certaines actions dans le logiciel.
La conception et la programmation orientées objet : On utilise une version simplifiée de la COO de G.Booch pour définir les classes et leurs relations en attendant une simplification pédagogique d'UML.

5.2 Programmez objet avec Delphi

Plan du chapitre:

1. Description générale de Delphi

*1.1 L'*application *Delphi*
1.2 Les fiches et les contrôles

2. Les modules dans Delphi

2.1 Partie " public " d'une UNIT : " Interface *"*
2.2 Partie " privée " d'une UNIT : " Implementation *"*
2.3 Initialisation et finalisation *d'une UNIT*

3. Delphi et la POO

3.1 Les éléments de base
3.2 Fonctionnalités
3.3 Les classes
3.3.1 Méta-classe
3.3.2 Classe amie
3.4 Le modèle objet
3.5 Les objets
3.6 Encapsulation
3.7 Héritage
3.8 Polymorphisme - *surcharge (bases)*
3.9 En résumé
3.10 Activité événementielle

4. Les propriétés en Delphi

4.1 Définition
4.2 Accès par read/write *aux données d'une propriété*
4.3 Propriétés tableaux
4.4 Surcharge *de propriété*

Introduction

Delphi™ de Borland est un RAD visuel fondé sur une extension orientée objet, visuelle et événementielle de Pascal, il fonctionne depuis 2004 sous le système Windows toutes versions, sous Linux et sous l'architecture .Net.

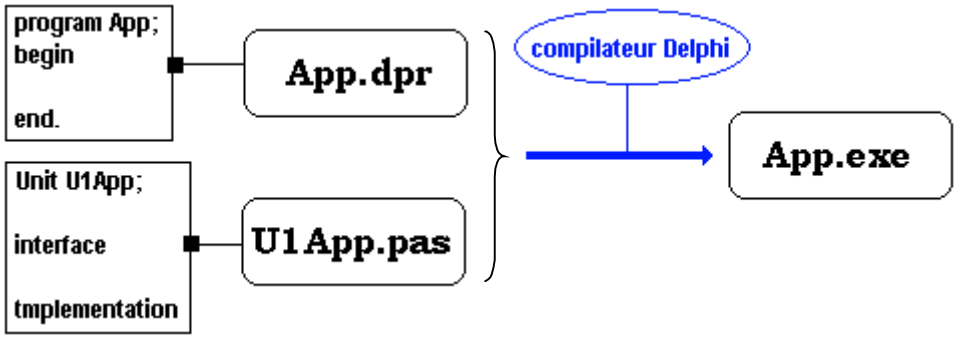
Pascal est le langage utilisé pour l'initiation dans de très nombreux établissements d'enseignement européens. Le RAD Delphi est un prolongement intéressant de ce langage. Nous allons explorer certains aspects les plus importants et significatifs du pascal objet de Delphi. Le langage pascal de base étant supposé connu par le lecteur, nous souhaitons utiliser ce RAD visuel en réutilisant du savoir faire pascal tout en y rajoutant les possibilités objet offertes par Delphi.

1. Description minimale de Delphi ...

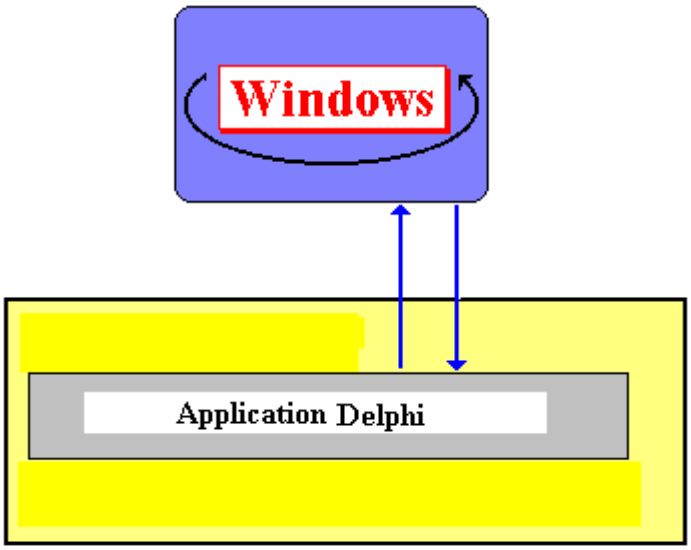
La version utilisée pour écrire les exemples est la dernière disponible sur Windows, mais tous les exemples sont écrits avec les fonctionnalités générales de Delphi ce qui permet de les compiler sur n'importe quelle version de Delphi depuis la version 5.

1.1 L'application Delphi

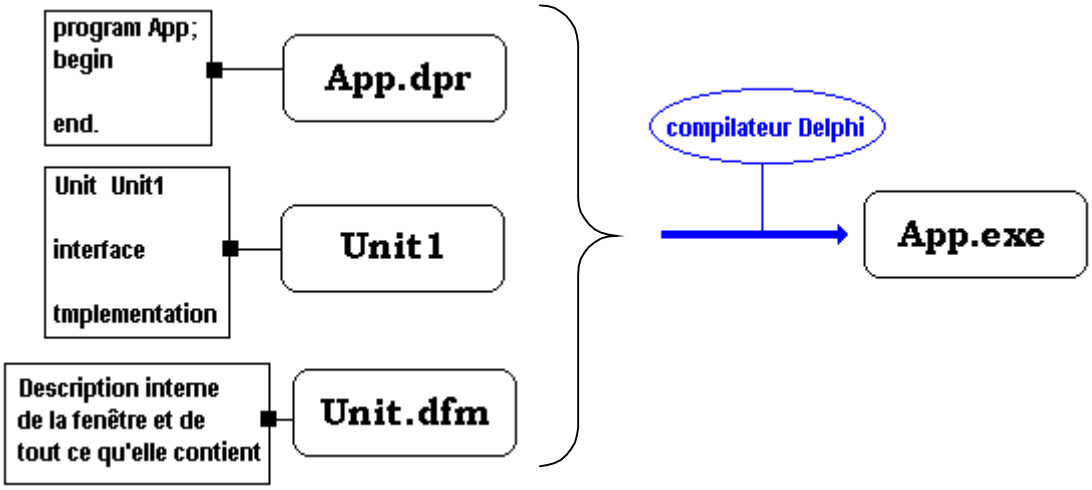
Une application console (non fenêtrée) Delphi se compose d'un projet "xxx.**dpr**" et d'au minimum un fichier d'Unit "xxx.**pas**" pour le code source. Lors de la compilation d'un projet Delphi engendre un code "xxx.**exe**" directement exécutable.

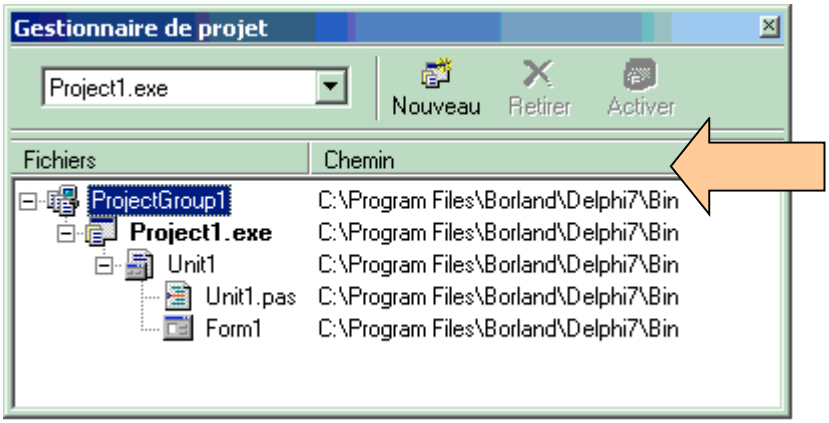


Tous les projets Delphi s'exécutent sous windows sans aucune Dll supplémentaire autre que celles que vous programmerez vous-même :



Une application fenêtrée Delphi se compose d'un projet "xxx.**dpr**" et d'au minimum deux fichiers de fiche "xxx.**dfm**" pour la description et "xxx.**pas**" et d'Unit pour le code source.





Ci-contre un projet minimal Delphi comportant une fiche principale.

Le projet se dénomme **Project1.dpr**

La fiche principale **Form1** et le code source de l'application sont rangés dans la Unit **Unit1.pas**.

La description de la fiche **Form1** et de ce qu'elle contient se trouve dans un fichier nommé **Unit1.dfm**.

A quoi sert une fiche ?

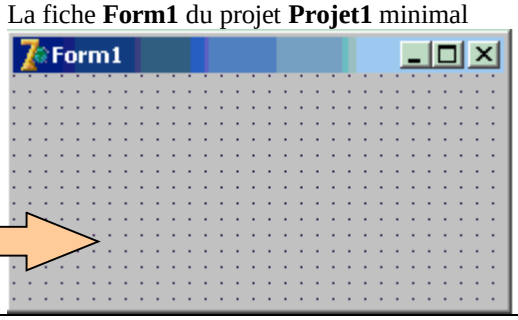
Les systèmes d'exploitation actuels sont dit fenêtrés au sens où ils fournissent un mode de communication avec l'homme fondé sur la notion de fenêtre, Windows en est un exemple.

La première action à entreprendre lors du développement d'une application **Interface Homme Machine** (IHM) avec un langage de programmation, est la création de l'interface de l'application et ensuite les interactions avec l'utilisateur. Le langage de programmation doit donc permettre de construire au moins une fenêtre

En Delphi pour créer une IHM, il faut utiliser des **fiches** (ou fenêtres) et des **contrôles**.

Ces fiches sont des objets au sens informatique de la POO, mais elles possèdent une **représentation visuelle**.

La fiche **Form1** du projet **Projet1** minimal

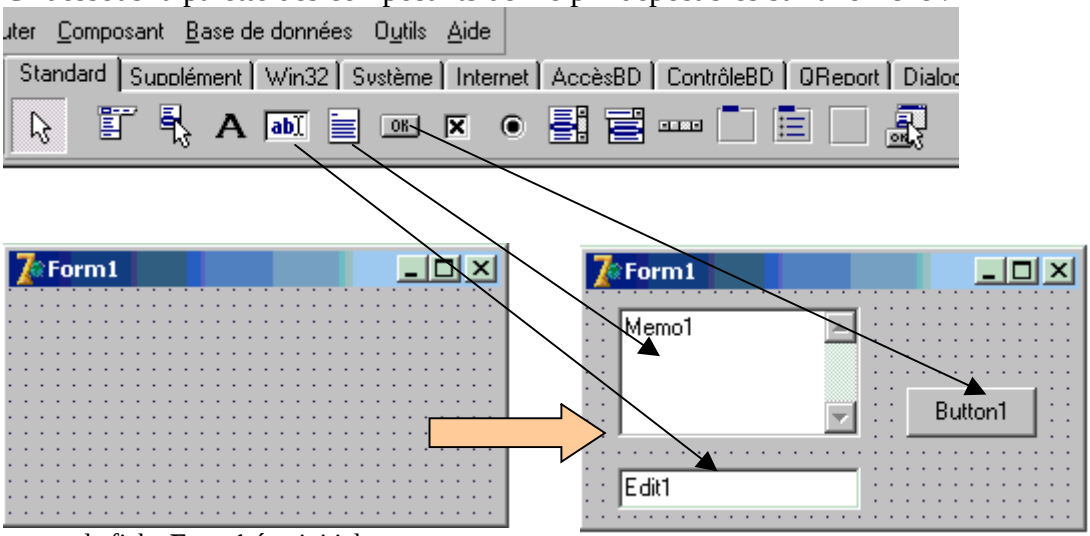


1.2 Les objets de fiches et les objets contrôles

Chaque fiche est en fait un objet instancié à partir de la classe interne des **TForm** de Delphi. Cette classe possède des propriétés(attributs) qui décrivent l'apparence de la fiche, des méthodes qui décrivent le comportement de la fiche et enfin des gestionnaires d'événements (pointeurs de méthodes) qui permettent de programmer la réaction de la fiche aux événements auxquels elles est sensible.

Sur une fiche vous déposez des **contrôles** qui sont eux aussi d'autres classes d'objets visuels, mais qui sont contenus dans la fiche.

Ci-dessous la palette des composants de Delphi déposables sur une fiche :



la fiche **Form1** état initial

la fiche **Form1** après dépôt de 3 contrôles

Que se passe-t-il lors du dépôt des contrôles ?

code dans la fiche **Form1** avant dépôt	code dans la fiche **Form1** après dépôt
unit Unit1; **interface** **uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs; **type** TForm1 = **class**(TForm) **private** *{ Déclarations privées }* **public** *{ Déclarations publiques }* **end**; **var** Form1: TForm1; **implementation** *{$R *.DFM}* **end**.	**unit** Unit1; **interface** **uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs; **type** TForm1 = **class**(TForm) Memo1: TMemo; Button1: TButton; Edit1: TEdit; **private** *{ Déclarations privées }* **public** *{ Déclarations publiques }* **end**; **var** Form1: TForm1; **implementation** *{$R *.DFM}* **end**.

Il existe dans Delphi une notion de classe conteneur, la fiche (classe **TForm**) en est le principal représentant. Delphi est un générateur automatique de programme source Pascal objet et dès l'ouverture du projet, il définit une classe conteneur **TForm1** qui hérite de la classe **TForm** et qui au départ ne contient rien :

```
TForm1 = class(TForm)
 private
   { Déclarations privées }
 public
   { Déclarations publiques }
 end;
```

Lorsque nous déposons les 3 contrôles Button1, Edit1, Memo1, ce sont des champs objets qui sont automatiquement ajoutés par Delphi dans le code source de la classe :

```
TForm1 = class(TForm)
  Memo1: T Memo;
  Button1: TButton;
  Edit1: TEdit;
 private
   { Déclarations privées }
 public
   { Déclarations publiques }
 end;
```

Donc l'environnement Delphi, n'est pas seulement un langage de programmation, mais aussi un générateur de programme à partir de dépôt de composants visuels, c'est la fonction d'un système RAD (Rapid Application Developpement).

Pour une utilisation de Delphi, nous renvoyons le lecteur à la documentation du constructeur, nous attachons par la suite à faire ressortir et à utiliser les éléments de Delphi qui concernent la programmation modulaire et la programmation objet

2. Les modules dans Delphi

Nous avons déjà vu au chapitre sur les TAD (types abstraits de données) que le Pascal de Delphi permettait de traduire sous une première forme la notion de type abstrait : la **Unit**. Une **Unit** Delphi permet aussi de représenter la notion de module.

- Une **Unit** est une unité compilable séparément de tout programme et stockable en bibliothèque.

- Une **Unit** comporte une partie " public " et une partie " privé ". Elle implante donc l'idée de module et étend la notion de bloc (procédure ou fonction) en Pascal. Elle peut contenir des descriptions de code simple ou de classe.

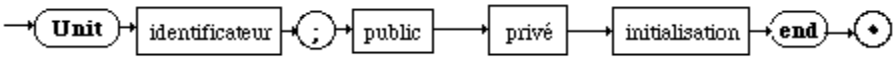
Chaque unité **Unit** est stockée dans un fichier **xxx.pas** distinct et compilée séparément ; les unités compilées (les fichiers xxx.**DCU**) sont liées pour créer une application.

Les unités en Delphi permettent aussi :

- De diviser de grands programmes en modules qui peuvent être modifiés séparément.
- De créer des bibliothèques qui peuvent être partagées par plusieurs programmes.
- De distribuer des bibliothèques à d'autres développeurs sans leur donner le code source.

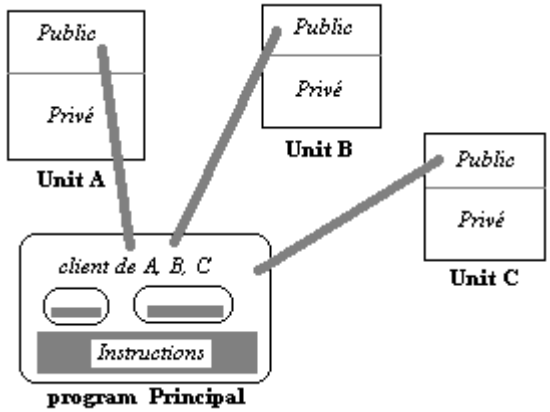
Pour générer un code éxécutable à partir d'un projet comportant plusieurs unités, le compilateur Delphi doit disposer, pour chaque unité, soit du fichier source xxx.**PAS**, soit du fichier XXX.**DCU** résultant d'une compilation antérieure. Cette dernière possibilité permet de fournir des unités compilées à d'autres personnes sans leur fournir le code source.

Syntaxe d'une unité :



Unit Truc;
<partie public >
<partie privée >
<initialisation >
end.

Utilisation d'une unité :



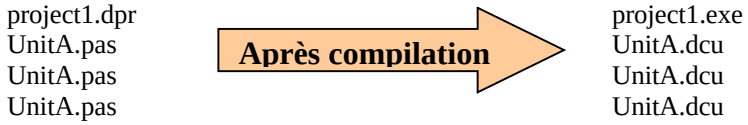
Le programme principal se nomme le projet, il est rangé dans le fichier "xxx.**dpr".**

Ici le programme principal utilise 3 Unit : UnitA , UnitB et UnitC

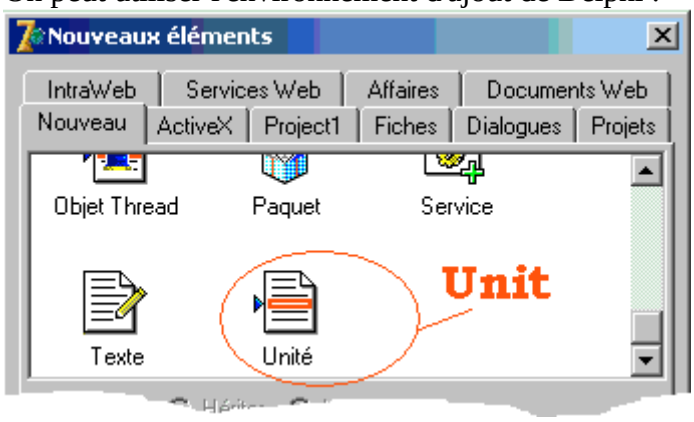
Squelette du code associé au schéma précédent :

unit UnitA; **interface** **implementation** **end**.	**unit** UnitB; **interface** **implementation** **end**.	**unit** UnitC; **interface** **implementation** **end**.	**Program** project1; **Uses** UnitA, UnitB, UnitC; **Begin** **end**.

Fichiers disque (sources et compilés) associés au code précédent :



Pour ajouter au projet une nouvelle **Unité** :

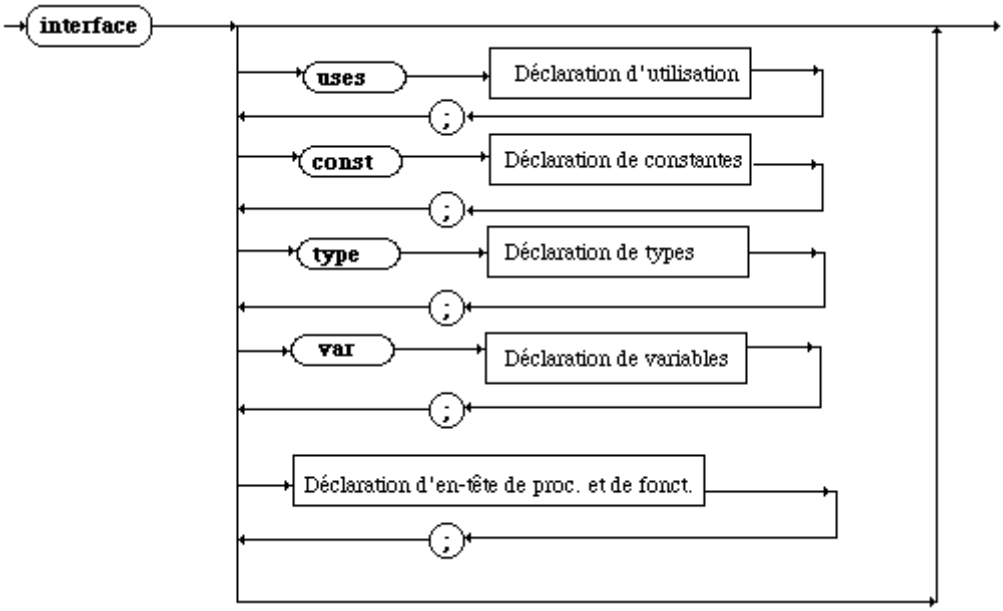
On peut utiliser l'environnement d'ajout de Delphi :  qui crée par exemple un fichier Unit2.pas contenant le squelette de la Unit2 et rajoute automatiquement cette **unit** au programme principal.	On peut écrire soi-même un fichier texte contenant la unit : **unit** Unit2; **interface** **implementation** **end**. Et rajouter soi-même au texte source du programme principal la **unit** : **Program** project1; **Uses** Unit2 ; **Begin** **end**.

2.1 Partie " public " d'une UNIT : " Interface "

Cette partie d'une **unit**, correspond exactement à la partie publique du module représenté par la UNIT. Cette partie décrit les en-têtes des procédures et des fonctions publiques et utilisables par les clients. Les clients peuvent être soit d'autres procédures Delphi, des programmes Delphi ou d'autres Unit.

La clause **Uses** XXX dans un programme Delphi, permet d'indiquer la référence à la **Unit** XXX et autorise l'accès aux procédures et fonctions publiques de l'interface dans tout le programme.

Syntaxe de l'interface :

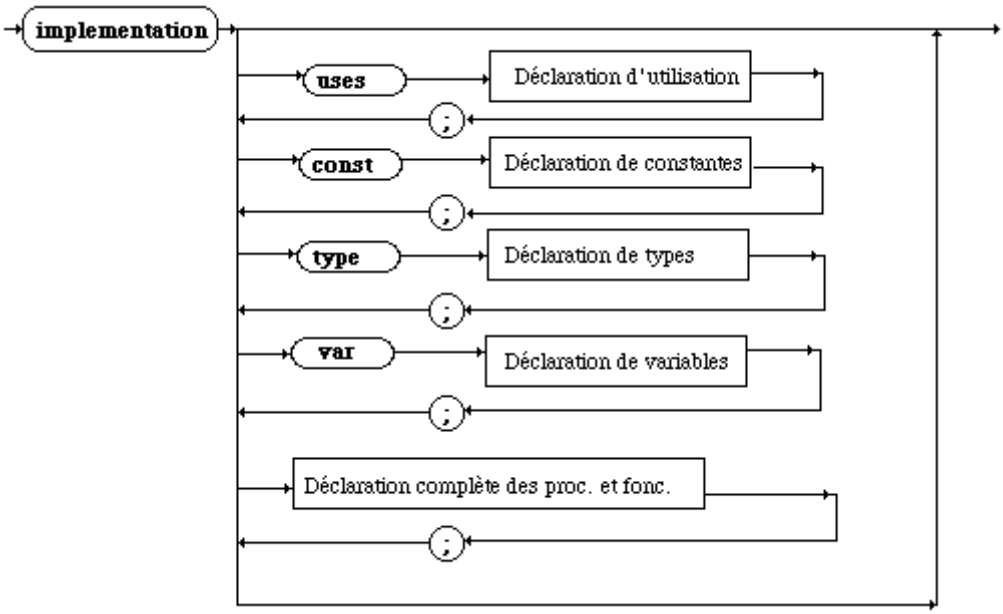


2.2 Partie " privée " d'une UNIT : " Implementation "

Correspond à la partie privée du module représenté par la UNIT. Cette partie intimement liée à l'interface, contient le code interne du module.

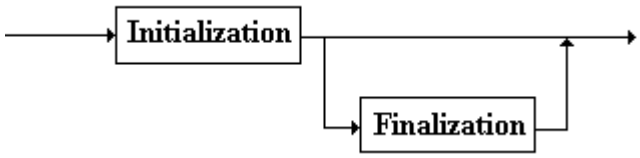
Elle contient deux sortes d'éléments : les déclarations complètes des procédures et des fonctions privées ainsi que les structures de données privées. Elle contient aussi les déclarations complètes des fonctions et procédures publiques dont les en-têtes sont présentes dans l'interface.

Syntaxe de l'implementation :



2.3 Initialisation et finalisation d'une UNIT

Syntaxe de l'initialisation :

La partie initialisation d'une unité en Delphi comporte deux sous parties **Initialization** et **Finalization** la seconde étant optionnelle:	

Initialization	Finalization
Il est possible d'initialiser des variables et d'exécuter des instructions au lancement de l'UNIT. Elles correspondent à des instructions classiques Pascal sur des données publiques ou privées de la Unit (initialisation de tableaux, mise à zéro de divers indicateurs, chargement de fichiers etc...).	Une fois que le code d'initialisation d'une unité a commencé à s'exécuter, la section de finalisation correspondante si elle existe, s'exécute obligatoirement à l'arrêt de l'application (libération de mémoire, de fichiers, récupération d'incidents etc...).

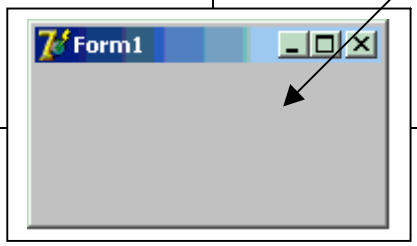
Exemple de programme console modulaire

Unit Delphi Uratio du TAD rationnel	Programme principal Delphi utilisant Uratio
unit Uratio**;** *{unité de rationnels spécification classique ZxZ/R}* **interface** **type** rationnel **=** **record** num**:** integer**;** denom**:** integer **end;** **procedure** reduire (**var** r**:** rationnel**);** **procedure** addratio (a, b**:** rationnel**; var** s**:** rationnel**);** **procedure** divratio (a, b**:** rationnel**; var** s**:** rationnel**);** **procedure** mulratio (a, b**:** rationnel**; var** s**:** rationnel**);** **procedure** affectQ(**var** s**:** rationnel**;** b**:** rationnel**);** **procedure** opposeQ(x**:**rationnel**;var** s**:**rationnel**);** **implementation** **procedure** reduire …. **procedure** addratio …. **Procedure** divratio …. **procedure** mulratio …. **procedure** affectQ…. **procedure** opposeQ…. **end.**	**program** essaiRatio**;** *{programme de test de la unit Uratio }* {$APPTYPE CONSOLE} **uses** SysUtils , Uratio**;** **var** r1, r2, r3, r4, r5**:** rationnel**;** **begin** r1.num **:=**18**;** r1.denom **:=** 15**;** r2.num **:=** 7**;** r2.denom **:=** 12**;** addratio(r1, r2, r3)**;** writeln('18/15 + 7/12 = ', r3.num, '/', r3.denom)**;** mulratio(r1, r2, r4)**;** writeln('18/15 * 7/12 = ', r4.num, '/', r4.denom)**;** divratio(r1, r2, r5)**;** writeln('18/15 / 7/12 = ', r5.num, '/', r5.denom)**;** r1.num **:=** 72**;** r1.denom **:=** 60**;** affectQ(r3,r1)**;** reduire(r1)**;** writeln('72/60 = ', r1.num, '/', r1.denom)**;** writeln('avant réduction ', r3.num, '/', r3.denom)**;** **end.**

Exemple fiche : le programme lançant une fiche vierge

```
unit Unit1;
    interface
    uses
      Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs;
    type
      TForm1 = class(TForm)
      private
        { Déclarations privées }
      public
        { Déclarations publiques }
      end;
    var
      Form1: TForm1;
    implementation
    {$R *.DFM}
end.
```

```
program Project1;

  uses  Forms,
    Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```



3. Delphi et la POO ...

Notre objectif est d'apprendre comment Delphi implante les notions contenues dans la P.O.O. et d'utiliser ces outils dans nos programmes.

Delphi est un langage orienté objet, dans ce domaine il possède des fonctionnalités équivalentes à C++ et java, avec sa version .Net, il possède les fonctionnalités équivalentes à celles de C# de microsoft.

3.1 Les éléments de base

Sachons que dans Delphi tout est objet, des contrôles de la **VCL** (Visual Component Library commportant plus de 600 classes dont environ 75 sont visuelles, les autres étant non visuelles ou concernant des objets de service).

Delphi possède de par son fondement sur Object Pascal tous les types prédéfinis du pascal et les constructeurs de types (supposés connus du lecteur), plus des types prédéfinis étendus spécifique à Delphi. Ce qui signifie qu'il est tout à fait possible de réutiliser en Delphi sans effort de conversion ni d'adaptation tout programme pascal ISO ou UCSD déjà écrit.

Les types **integer**, **real** et **string** sont des types génériques c'est à dire qu'ils s'adaptent à tous les types d'**entiers**, de **réels** et de **chaînes** de Delphi.

Par exemple les entiers de base de Delphi suivants :

Shortint	-128..127	8 bits signé
Smallint	-32768..32767	16 bits signé
Longint	-2147483648..2147483647	32 bits signé
Byte	0..255	8 bits non signé
Word	0..65535	16 bits non signé
Longword	0..4294967295	32 bits non signé

peuvent être affectés à une variable x : **integer** car c'est un type générique.

var x : **integer** ;
a : Longint; b : Longword; c: Byte; d: Word; e: Smallint; f: Shortint;
x := a ; x := b ; x := c ; x := d ; x := e ; x := f ;

Delphi dispose d'un type générique **variant polymorphe** sur les types de données prédéfinis de Delphi. Un **variant** peut s'adapter ou se changer en pratiquement n'importe quel type de Delphi	**Var** x : variant; **begin** x:='abcdefgh'; *// x est une string* x:=123.45; *// x est un real* x:=true; *// x est un booléen* x:=5876 *// x est un integer* etc...

paramètres

Les passages des paramètres s'effectuent principalement selon les deux modes classiques du pascal : le passage par valeur (pas de mot clé), le passage par référence (mot clé **Var**). Par défaut si rien n'est spécifié le passage est par valeur.

Améliorations de ces passages de paramètres :

- Delphi autorise un mode de passage dit des paramètres constants permettant une sécurité accrue au passage par valeur (mot clé **Const**).
- Delphi dispose d'une autre amélioration concernant le passage par référence : le passage par sortie (mot clé **out**), qui indique simplement à la procédure où placer la valeur en sortie sans spécifier de valeur en entrée, qui si elle existe n'est pas utilisée.

Exemple d'utilisation des variant dans un tri récursif sur un tableau de variant :

(les paramètres sont tous passés par référence)

```
const n=100;
type Tableau =array[0..n] of variant;

procedure quicksort (var G,D:integer; var
tabl:Tableau);
var i , j : Integer;
   x , w : variant;
begin
 i := G;
 j := D;
 x := tabl[(i + j) div 2];
 repeat
   While tabl[i] < x do  i := i + 1;
   While tabl[j] x do  j := j - 1;
   If i <= j Then
   begin
      w := tabl[i];
      tabl[i] := tabl[j];
      tabl[j] := w;
      i := i + 1;
      j := j - 1
   End;
 Until i j;
 If G < j Then quicksort(G, j, tabl);
 If D i Then quicksort(i, D, tabl)
End;
```

La procédure générique quicksort permet de trier un tableau de **variant**.

Grâce à son pouvoir polymorphe un **variant** peut devenir au choix soit:

Entier long
Entier court
Un réel simple ou double
Une chaîne de caractères.

Ce qui revient à dire que la procédure générique quicksort permet de trier un tableau de :

Entiers longs
Entiers courts
Réels simples ou doubles
Chaînes de caractères.

Dans le cas où le type **variant** n'existe pas, il faut au moins 3 procédures différentes quicksortXXX qui diffèrent par le type de leur paramètres, mais avec le même code :

procedure quicksortEntier (sur un tableau d'**integer**)
procedure quicksortReel (sur un tableau de **real**)
procedure quicksortString (sur un tableau de **string**)

Remarque :
D'autres modes de passage des paramètres sont possibles en Delphi, nous n'en parlons pas ici.

3.2 Fonctionnalités du pascal objet de Delphi

Delphi est un langage à structure de blocs complet. Sa gestion est identique à celle du langage pascal.

- La visibilité est identique à celle qui est inhérente à tout langage de bloc Algol-like : variables globales, variables locales, masquage des identificateurs.

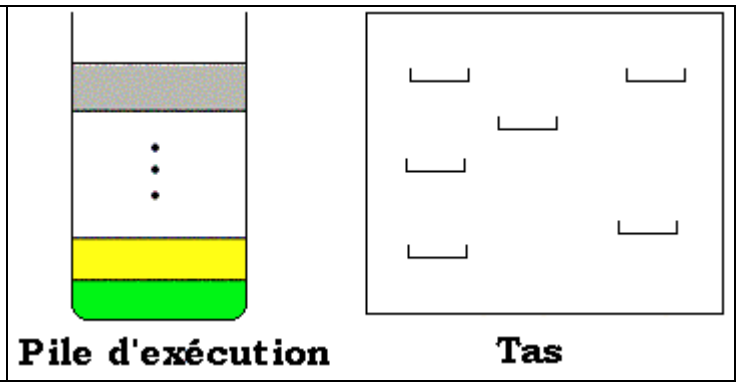
- Les entités gérées dynamiquement le sont selon un modèle classique de **tas** et de **pile d'exécution**.

- Les entités gérées d'une façon permanente (données globales)sont persistantes durant toute la durée de l'exécution du programme, elles existent dans le **segment de données** alloué à l'application.

Dans un langage à structure de bloc comme Delphi, la mémoire centrale contient deux entités fondamentales : le **tas** et de **pile d'exécution**.

- Le **tas** est une structure de données déjà étudiée (arbre parfait partiellement ordonné géré dans un tableau).

- La **pile d'exécution** est une pile LIFO.

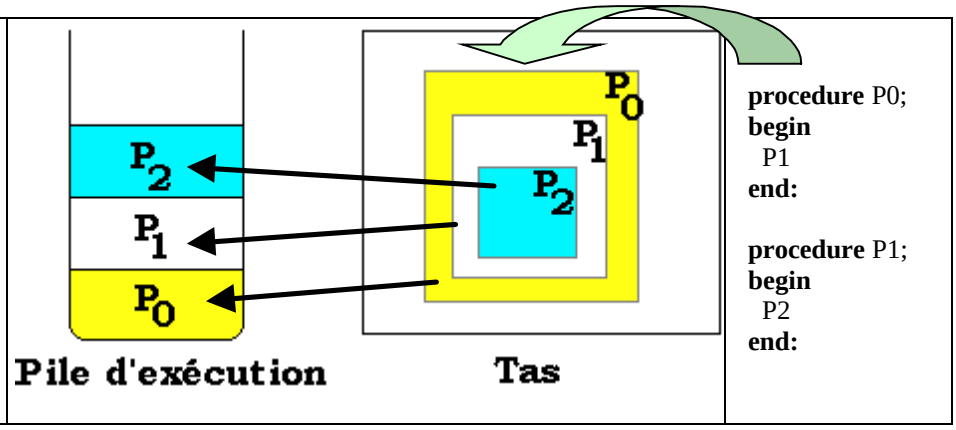


La pile d'exécution de Delphi a une taille paramétrable par le programmeur (maximum=2 147 483 647 octets), elle permet toutes les récursivités utiles.

Ci-dessous l'illustration du fonctionnement du **tas** et de la **pile** dans le cas de l'appel d'une procédure P0 qui appelle une procédure P1 qui appelle elle-même une procédure P2:

- Le **tas** contient les structures dynamiques et les codes.

- La **pile d'exécution** contient les contextes des procédures appelées.



Delphi est un langage acceptant la récursisvité

La récursivité d'une procédure ou d'une fonction est la capacité que cette fonction/procédure a de s'appeler elle-même.

Soit par exemple la somme des n premiers entiers définie par la suite récurrente S_n :

$$S_n = S_{n-1} + n$$
$$S_0 = 0$$

Ci-dessous une fonction de calcul récursif de la somme des **n** premiers entiers :

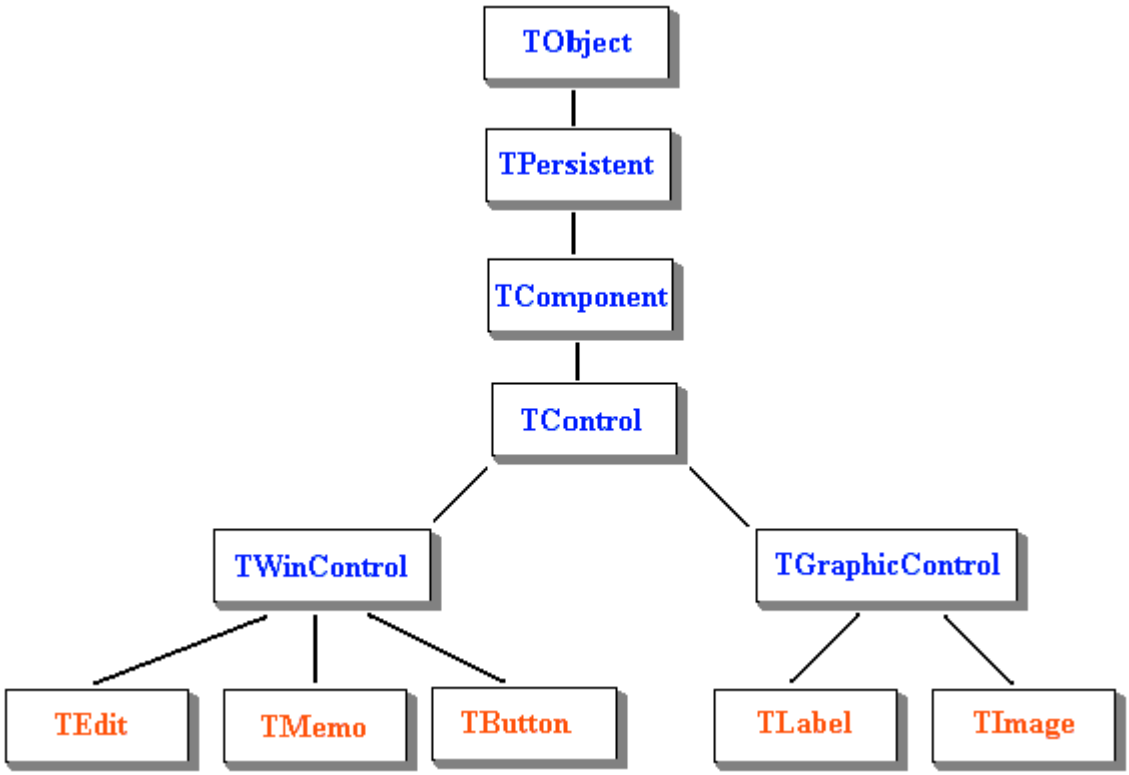
```
Function S(n : integer) : integer;
begin
 If n = 0 Then
    result:= 0
 Else
    result := n + S(n - 1)
End;
```

Delphi affiche un message d'erreur de pile pleine sur l'appel **S(129937)**, donc une profondeur de 129936 appels recursifs a été atteinte sur ce programme avec **le paramètrage standard fournit de 1 Mo comme taille maximum de la pile**.

3.3 Les classes

La notion de *classe* est essentielle dans Delphi en mode application visuelle. Les classes sont déclarées comme des types et contiennent des champs, des méthodes et des propriétés. Il existe une classe d'objet primitive appelée **TObject** , qui est l'ancêtre de toutes les autres classes de Delphi.

Voici un extrait de la hiérarchie des classes visuelles VCL (Visual Component Library) dans Delphi :



Comment déclarer une classe en Delphi

Un type classe doit être déclaré et nommé avant de pouvoir être instancié. Une classe est considéré par Delphi comme un nouveau type et doit donc être déclaré là où en Pascal l'on construit les nouveaux types : au paragraphe des déclarations à l'alinéa de déclaration des types.

Les types classes peuvent être déclarés pratiquement partout où le mot clé type est autorisé à l'exception suivante : ils ne doivent pas être déclarés à l'intérieur d'une procédure ou d'une fonction.

Les deux déclarations ci-dessous sont équivalentes :

Type ma_classe = **class** {déclarations de champs } {spécification méthodes } {spécification propriétés } **end**;	**Type** ma_classe = **class**(Tobject) {déclarations de champs } {spécification méthodes } {spécification propriétés } **end**;

Méta-classe

Delphi autorise la construction de méta-classes. Une Méta-classe est un générateur de classe. En Delphi les méta-classes sont utilisées afin de pouvoir passer des paramètres dont la valeur est une classe dans des procédures ou des fonction.

Les méta-classes en Delphi sont représentées par une référence de classe

Une méta-classe	Une variable de méta-classe
Type TMetaWinClasse = **class of** TWinControl;	**var** x : TMetaWinClasse;

La variable x de classe TMetaWinClasse, peut contenir une référence sur n'importe quelle classe de TWinControl : x :=Tmemo; x :=TEdit; x :=TButton; …

Exemple d'utilisation de la notion de méta-classe pour tester le type réel du paramètre effectif lors de l'appel de TwinEssai.	**procedure** TwinEssai (UnWinControl : TmetaWinClasse); **begin** **if** UnWinControl =TEdit **then …** **else if** UnWinControl=TMemo **then …** ....etc **end**;
A comparer avec l'utilisation de l'opérateur is sur le même problème	**procedure** TwinEssai (UnWinControl : TWinControl); **begin** **if** UnWinControl **is** TEdit **then …** **else if** UnWinContro **is** TMemo **then …** ....etc **end**;

En passant à un constructeur un paramètre de méta-classe on peut alors construire des objets dont la classe ne sera connue que lors de l'exécution.

Où déclarer une classe en Delphi ?

Amies

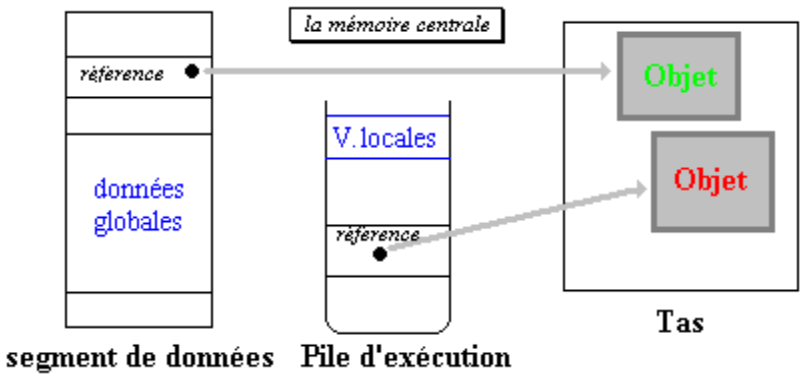
- Les classes sont déclarées dans des **unit**.
- Toutes les classes déclarées dans la même **unit** sont **amies**, c'est à dire que chacune d'elle a accès aux membres **privés** de toutes les autres classes declarées dans cette **unit**.
- Si l'on souhaite avoir **n** classes **non amies**, il faut les déclarer chacune dans une **unit** séparée.

3.4 Le modèle objet de Delphi

Le modèle physique choisi pour Delphi est celui de la référence :

- Chaque objet est caractérisé par un couple (référence, bloc de données).
- Si la variable de référence est locale à une procédure,elle est alors située physiquement dans la **pile d'exécution** avec les autres variables locales.
- Si la variable de référence est globale, elle est située physiquement dans le **segment de données** (permanent durant toutes la durée de l'exécution). Dans les deux cas, le bloc de données (l'objet effectif) est alloué dans le **tas**.

Ci-dessous une carte mémoire fictive d'un processus (une application Delphi classique à un seul thread):



La simplicité du modèle (semblable aux variables dynamiques ou pointeurs du pascal) permet de dire qu'en Delphi les pointeurs sont sous-jacents mais entièrement encapsulés.

Ce modèle impose une contrainte d'écriture en découplant la déclaration d'objet et sa création.

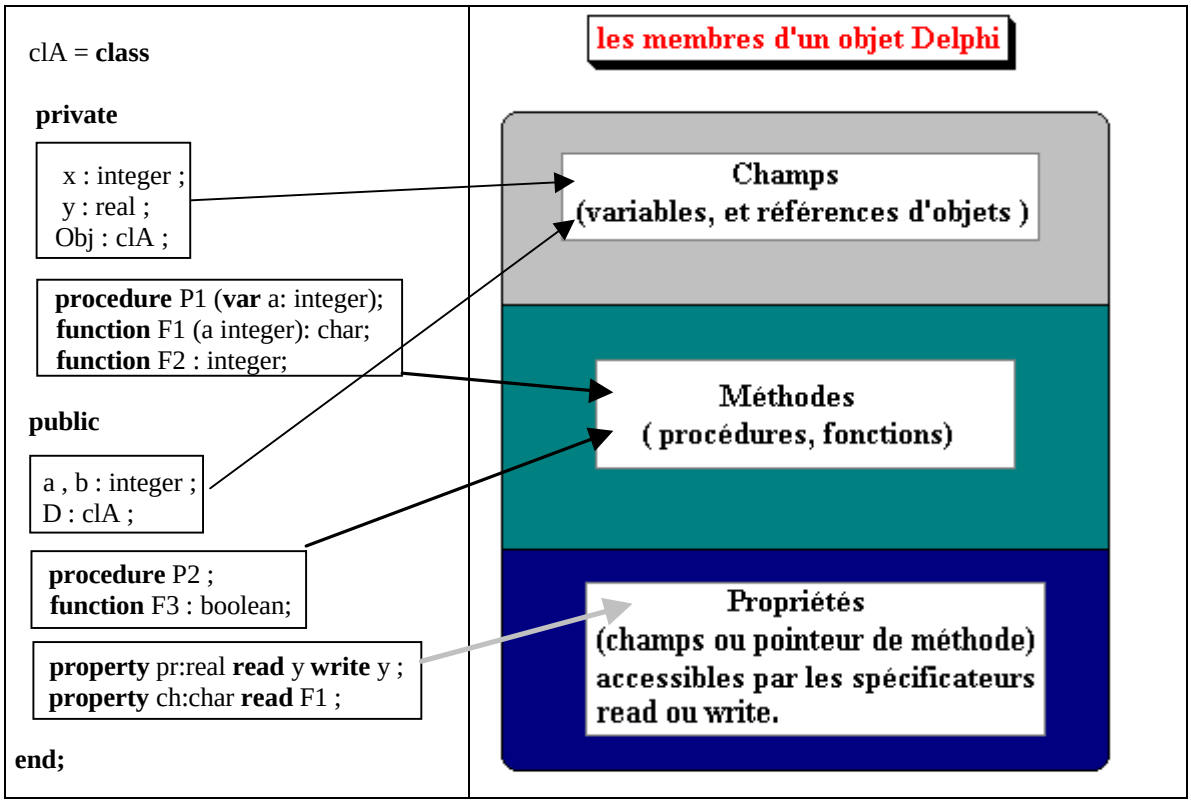
3.5 Les objets en Delphi

Un *objet* Delphi suit très exactement les définitions générales d'objets.

Un *objet* Delphi est une *instance* d'une classe.
Un *objet* Delphi contient des membres qui sont :

- des variables pascal (*champs ou attributs*)
- des procédures et des fonctions (*méthodes*)
- des **propriétés**.

Nous verrons plus loin que ces propriétés peuvent être de différentes catégories, en première lecture nous dirons que ce sont des champs possédant des spécificateurs d'accès.



Les classes et les objets sont déclarés dans les unités (dans la partie déclaration de l'**interface** ou de l'**implementation** de l'unité), le code des méthodes est défini uniquement dans la partie **implementation** de l'unité.

Constructeur / Destructeurs d'objets

En Delphi, chaque variable d'instance (*objet instancié*) doit obligatoirement être initialisée et peut être détruite lorsqu'elle n'est plus utile. Tout objet doit être d'abord construit avant son utilisation. Ceci se fait à l'aide d'une méthode spécifique déclarée par le mot clef **constructor**.

La destruction d'une instance s'effectue par une méthode aux propriétés identiques à un **constructor** et elle se dénomme un **destructor**.

- Par défaut les classes disposent (provenant de la classe mère **TObject**) d'une méthode permettant la construction des objets de la classe : cette méthode de type **constructor** est dénommée **Create**. Ce genre de méthode assure la création de l'objet physique en mémoire et sa liaison avec l'identificateur déclaré dans le code (l'identificateur contient après l'action du Create, l'adresse physique de l'objet).

- Il en est de même pour la désallocation des objets de vos classes (leur destruction), celles-ci disposent d'un destructeur d'objets dénommé **Destroy**. Cette méthode de type **destructor** rend au processus, tout l'espace mémoire utilisé par l'objet physique, mais attention elle ne dé-référence pas l'identificateur, si nécessaire cette opération est à la charge du programmeur (grâce à l'instruction : **identificateur:=nil**).

Unit Uclasses contenant une classe clA	Programme principal utilisant Uclasses

Unit Uclasses ;
Interface type
clA = **class**
private
x : integer ;
y : real ;
Obj : clA ;
procedure P1 (**var** a: integer);
function F1 (a integer): char;
function F2 : integer;
public
a , b : integer ;
D : clA ;
procedure P2 ;
function F3 : integer;
property prop1: integer **read** x **write** x ;
property prop2 : char **read** F1 ;
end;
 	
implementation
procedure clA.P1 (**var** a: integer); …
function clA.F1 (a integer): char; …
function clA.F2 : integer; …
procedure clA.P2 ; …
function clA.F3 : boolean; …
end.
	Program principal ;
	
	Uses Uclasses ;
	
	procedure utilise (ObjA : clA);
	begin
	if ObjA.a > 5 **then**
	begin
	ObjA.P2 ;
	ObjA.b := ObjA.F3 - ObjA.a +1;
	end
	else
	prop1 := ObjA.a -1
	end;
	
	var refA : clA ;
	
	begin
	refA : = clA.Create ;
	utilise (refA) ;
	refA.destroy;
	end.

Déclaration

code des méthodes

Déclaration d'une référence de type clA.

Instanciation d'un objet de type clA.

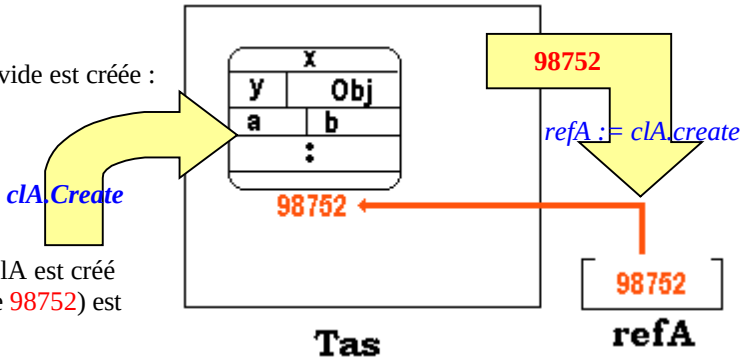
Destruction de l'objet de type clA.

var refA : clA ;

Lors de la déclaration une variable **refA** vide est créée :

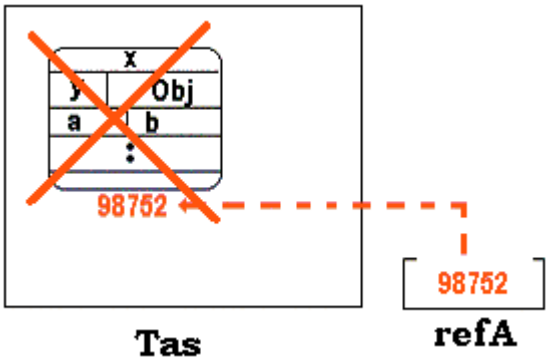
refA : = clA.Create ;

Lors de l'instanciation, un objet de type clA est créé dans le tas, puis sa référence (son adresse 98752) est mise dans la variable **refA** :



refA.destroy;

Lors de la destruction , l'objet du tas est détruit, la mémoire qu'il occupait est rendu au tas et la variable refA ne référence plus un objet.

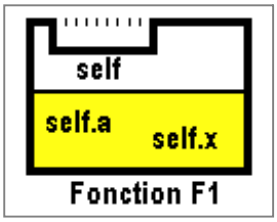


Le paramètre implicite self

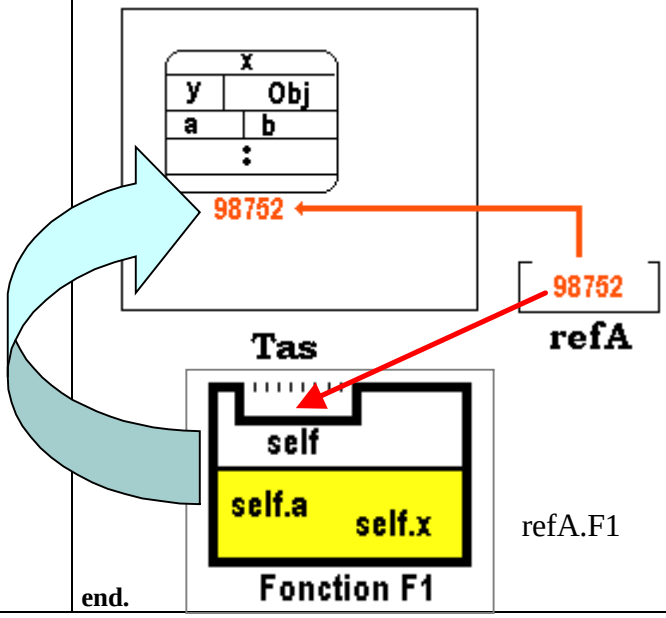
- Etant donnée une classe quelconque, chaque méthode de cette classe possède systématiquement dans son implémentation un paramètre implicite dénommé **Self**, que vous pouvez utiliser.
- Cet identificateur **Self** désigne l'objet (lorsqu'il sera instancié)dans lequel la méthode est appelée.

Exemple:

```
Unit Uclasses;
Interface
type
clA = class
  private
    Obj : clA ;
 public
   x , y : integer ;
   a , b : integer ;
  function F1 : integer;
end;

implementation

function clA.F1 : integer;
begin
 self.a := self.x +2;
 result := self.a;
end;

end.
```




```
Program principal ;

  Uses Uclasses ;
var refA : clA ;
    n : integer;

  begin
   refA : = clA.Create ; ….
```




```
end.
```

Lors de l'instanciation (refA : = clA.Create) la valeur 98752 de la référence, est passée automatiquement dans la variable implicite **self** de chaque méthode de l'objet **refA**, ce qui a pour résultat que dans la méthode F1, les expressions formelles **self.a** et **self.x** prennent une valeur effective et désignent les valeurs effectives des champs **a** et **x** de l'objet n° **98752** du **tas**.

3.6 Encapsulation

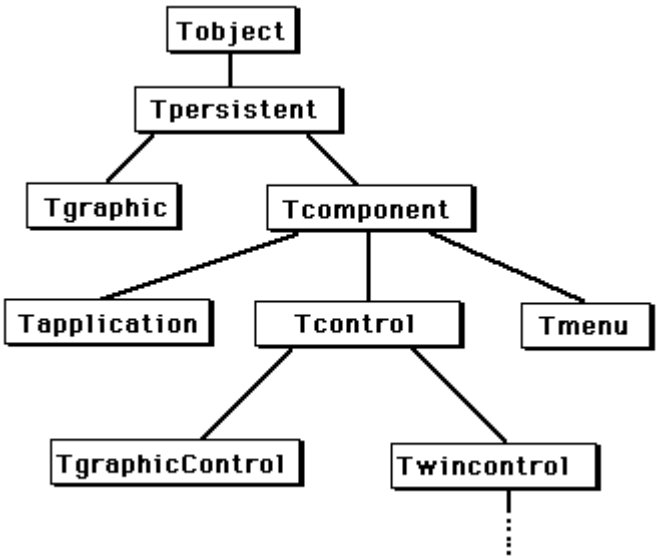
Rappelons que pratiquement, en POO l'encapsulation permet de masquer les informations et les opérations d'un objet aux autres objets. Contrairement à certains autres langages orientés objet, dans Delphi par défaut, s'il n'y a pas de descripteur d'encapsulation, tout est visible (donc **public**).

Delphi possède au moins quatre niveaux d'encapsulation des informations dans un objet qui sont matérialisés par des descripteurs :

published : Les informations sont accessibles par toutes les instances de toutes les classes (les *clients*) + accessibles à l'inspecteur d'objet de Delphi.
public : Les informations sont accessibles par toutes les instances de toutes les classes (les *clients*).
protected : Les informations ne sont accessibles qu'à toutes les instances de la classe elle-même et à toutes celles qui en *héritent* (ses descendants).
private : Les informations ne sont accessibles qu'à toutes les instances de la classe elle-même.

3.7 Héritage

Il s'agit en Delphi de *l'héritage simple* (graphe arborescent) dans lequel une famille dérive d'une seule classe de base. Voici une partie du graphe d'héritage simple de certaines classes de Delphi :



Syntaxe de la déclaration d'héritage :

Type classe_fille = **class**(classe_ancetre)

Type classe_ancetre = **class** { champs } { méthodes } **end**;	**Type** classe_fille = **class**(classe_ancetre) { champs } { méthodes } **end**;

Nous indiquons donc ainsi en Delphi que la classe_fille hérite de la classe_ancetre.

Nous avons vu que les deux déclarations ci-dessous étaient équivalentes :

Type ma_classe = **class** {déclarations de champs } {spécification méthodes } {spécification propriétés } **end**;	**Type** ma_classe = **class**(TObject) {déclarations de champs } {spécification méthodes } {spécification propriétés } **end**;

L'écriture de gauche indique en fait que toutes les classes déclarées sans qualificatif d'héritage héritent **automatiquement** de la classe TObject. La VCL de Delphi est entièrement construite par héritage (une hiérarchie objet complète) à partir de TObject.

*Ci-dessous la déclaration de la classe **TObject** dans Delphi :*

```
TObject = class
  constructor Create;
  procedure Free;
  class function InitInstance(Instance: Pointer): TObject;
  procedure CleanupInstance;
  function ClassType: TClass;
  class function ClassName: ShortString;
  class function ClassNameIs(const Name: string): Boolean;
  class function ClassParent: TClass;
  class function ClassInfo: Pointer;
  class function InstanceSize: Longint;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function MethodAddress(const Name: ShortString): Pointer;
  class function MethodName(Address: Pointer): ShortString;
  function FieldAddress(const Name: ShortString): Pointer;
  function GetInterface(const IID: TGUID; out Obj): Boolean;
  class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
  class function GetInterfaceTable: PInterfaceTable;
  function SafeCallException(ExceptObject: TObject;
    ExceptAddr: Pointer): HResult; virtual;
  procedure AfterConstruction; virtual;
  procedure BeforeDestruction; virtual;
  procedure Dispatch(var Message); virtual;
  procedure DefaultHandler(var Message); virtual;
  class function NewInstance: TObject; virtual;
  procedure FreeInstance; virtual;
  destructor Destroy; virtual;
 end;
```

*La classe Tobject utilise ici la notion de **méthode de classe** :*

- Une méthode de classe est une méthode (autre qu'un constructeur) qui **agit sur des classes** et non sur des objets.

- La définition d'une méthode de classe doit commencer par le mot réservé **class.**

Par exemple dans Tobject :

```
TObject = class
  ...
  class function ClassName: ShortString;
  class function ClassParent: TClass; ...
```

- La déclaration de définition d'une méthode de classe doit également commencer par class :

```
class function TObject.ClassName: ShortString;
begin
  //...le paramètre self est ici la classe Tobject
end;
```

etc...

Outre la classe TObject, Delphi fournit le type de méta-classe (référence de classe) générale **TClass** :

```
TClass = class of TObject;
```

3.8 surcharge de méthode

Signature d'une méthode -- définition C'est son nom , le nombre et le type de ses paramètres

On dit qu'une méthode est surchargée dans sa classe si l'on peut trouver dans la classe plusieurs signatures différentes de la même méthode.

En Delphi, les en-têtes des méthodes surchargées de la même classe, doivent être suivies du qualificateur **overload** afin d'indiquer au compilateur qu'il s'agit d'une autre signature de la même méthode.

Dans la classe classeA nous avons déclaré 3 surcharges de la même méthode Prix, qui pourrait évaluer un prix en fonction du montant rentré selon trois types de données exprimées en yen, en euro ou en dollar.	**Type** classeA = **class** **public** **function** Prix(x:yen) : real;**overload**; **function** Prix(x:euro): real;**overload**; **function** Prix(x:dollar) : real;**overload**; **end**;

Comment appeler une surcharge de méthode ?

Soit le programme de droite qui utilise la classeA définie à gauche avec 3 surcharges de la méthode Prix

```
Unit Uclasses ;
interface

Type
yen = class … end;
euro = class … end;
dollar = class … end;
classeA = class
    public
        function Prix(x:yen) : real;overload;
        function Prix(x:euro): real;overload;
        function Prix(x:dollar) : real;overload;
 end;

implementation

function classeA.Prix(x:yen) : real; // signature n°1
begin
  …
end;

function classeA.Prix(x:euro): real;
begin
  …
end;

function classeA.Prix(x:dollar) : real;
begin
  …
end;
```

```
Program principal ;
Uses Uclasses ;

var refA : classeA;
      a : yen ; b : euro ; c : dollar ;

Procedure Calcul1 (ObjA : classeA; valeur : yen );
begin
  ObjA.Prix ( valeur)
end;

Procedure Calcul2 (ObjA : classeA;  valeur : euro);
begin
  ObjA.Prix ( valeur )
end;

Procedure Calcul3 (ObjA : classeA;  valeur : dollar);
Var ObjA : classeA ;
begin
  ObjA.Prix ( valeur)
end;

begin
 refA:= classeA.Create ;
 a := yen.Create ;
 b := euro.Create ;
 c := dollar.Create ;
 Calcul1 ( refA , a );
 Calcul2 ( refA , b );
 Calcul3 ( refA , c );
End.
```

Le compilateur connaît le type du second paramètre, lorsqu'il appelle :
ObjA.Prix (valeur)

Il va alors chercher s'il existe une signature correspondant à ce type et va exécuter le code de la surcharge adéquate

- L'appel ObjA.Prix(valeur) dans Calcul1(refA , a) sur a de type yen provoque la recherche de la signature suivante **Prix** (type **yen**), c'est la *signature n°1* qui est trouvée et exécutée.

- L'appel ObjA.Prix(valeur) dans Calcul2(refA , b) sur a de type euro provoque la recherche de la signature suivante **Prix** (type **euro**)), c'est la *signature n° 2* qui est trouvée et exécutée.

- L'appel ObjA.Prix(valeur) dans Calcul3(refA , c) sur a de type dollar provoque la recherche de la signature suivante **Prix** (type **dollar**)), c'est la *signature n°3* qui est trouvée et exécutée.

4. Les propriétés en Delphi

Une propriété définie dans une classe permet d'accéder à certaines informations contenues dans les objets instanciés à partir de cette classe. Une propriété a la même syntaxe de définition et d'utilisation que celle d'un champ d'objet (elle possède un type de déclaration), mais en fait elle peut invoquer une ou deux méthodes internes pour fonctionner ou se référer directement à un champ. Les méthodes internes sont déclarées à l'intérieur d'un bloc de défintion de la propriété.

Nous nous limiterons aux propriétés non tableau, car cette notion est commune à d'autres langages (C# en particulier)

4.1. Définition

Comme un champ, une propriété définit un attribut d'un objet.

Mais un champ n'est qu'un emplacement de stockage dont le contenu peut être consulté et modifié, tandis qu'une propriété peut associer des actions spécifiques à la lecture et lors de la modification de ses données : une propriété peut être utilisée comme un attribut.

Les propriétés proposent un moyen de contrôler l'accès aux attributs d'un objet et autorisent ou non le calcul sur les attributs.

Syntaxe :

property nomPropriété[indices] : type [index constanteEntière] spécificateurs ;

Remarque :

il faut au minimum un descripteur (ou spécificateur) d'accès pour une propriété :

- soit lecture seule (spécificateur **read**),
- soit écriture seule (spécificateur **write**),
- soit lecture et écriture **read …. write** .

Attention :

Une propriété **ne peut pas être transmise comme paramètre** référence dans une procédure ou une méthode !

Exemple de syntaxe d'écriture de propriétés :

```
classeA = class
  public
    property prop1 : integer  read y write z ; // lecture et écriture
    property prop2 : char read F1 ; // lecture seule
    property prop3 : string write t ; // écriture seule
 end;
```

4.2. Accès par read/write aux données d'une propriété

Après les spécificateurs **read** et **write**, il obligatoire de préciser le moyen d'accès à la propriété : Ce moyen peut être un **attribut**, ou une **méthode**.

Accès à une propriété par un attribut

property *propriété1: type* **read** *Fpropriété1* ;

La **property** *propriété1* fait référence à l'attribut *Fpropriété1* et en permet l'accès en lecture seule.

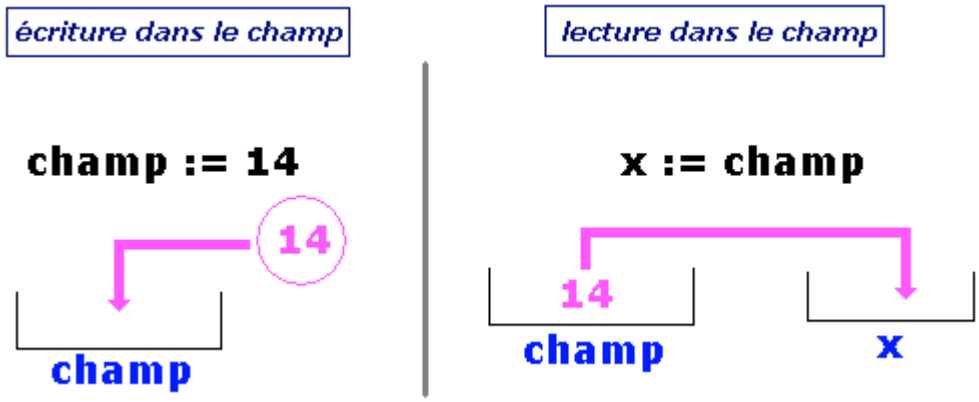
Pour avoir un intérêt, la **property** *propriété1* doit être déclarée en **public**, tandis que l'attribut *Fpropriété1* est déclaré en **private**.

Lorsque la propriété est définie, il faut que le champ auquel elle se réfère ait déjà été défini dans la classe, ou dans une classe ancêtre.

D'une façon générale on peut comparer la lecture et l'écriture dans un champ et dans une propriété comme ci-dessous :

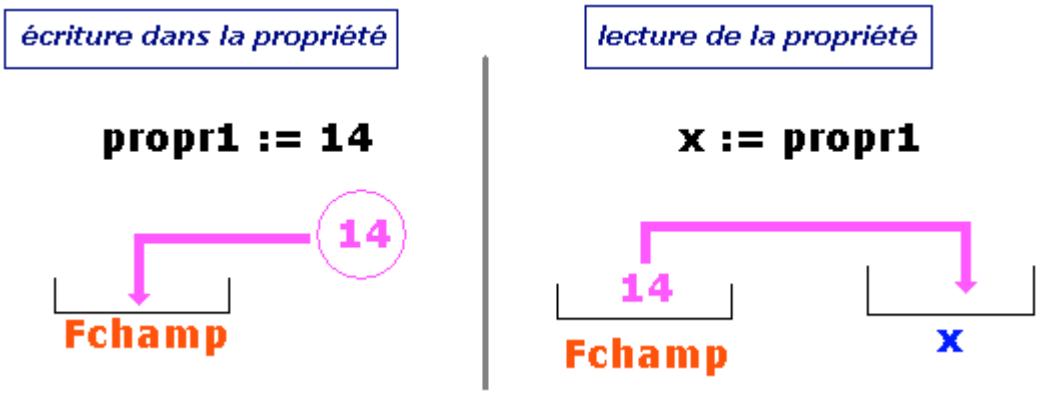
Soit la classe classeA :

```
classeA = class
   public
     champ : integer ;
 end;
```



Soit une autre version de la classe classeA :

```
classeA = class
   private
     Fchamp : integer ;
   public
     property propr1 : integer read Fchamp write Fchamp ;
 end;
```



On peut donc assimiler la propriété **propr1** à une clef ouvrant une porte sur un champ privé de l'objet. et autorisant la lecture et/ou l'écriture dans ce champ.

Exemple d'utilisation :

```
Unit1.pas
Unit1
type
  MaClasse = class
  private
    Fprop1 : Integer;
    //déclaration privée d'un attribut
  public
    Property prop1 : Integer Read Fpropr1 Write Fprop1 ;
    // déclaration publique d'une propriété permettant l'accés
    // à l'attribut Fprop1 en lecture et écriture
  end;

var
  MonObjet: MaClasse;

implementation

Procedure Quelconque();
begin
  ...
  MonObjet := MaClasse.Create;
  ...
  MonObjet.prop1 := -12;       // accés en écriture
  i := MonObjet.prop1;         // accés en lecture
  ...
end;
22: 49   Modifié   Insertion
```

Accès à une propriété par une méthode

Cas du spécificateur d'accès **read**

```
Unit1.pas
Unit1
type
  MaClasse = class
  private
    Function LireProp1 : TColor;
    //déclaration privée d'une méthode d'accés
  public
    Property Prop1 : TColor Read LireProp1 ;
    // déclaration publique d'une propriété permettant l'accés
    // à la méthode LireProp1 en lecture Seule
  end;
16: 47   Modifié   Insertion
```

La méthode d'accès en lecture **LireProp1** doit :

- être une fonction,
- être sans paramètres,
- renvoyer un résultat du même type que celui de la propriété.

Cas du spécificateur d'accès **write**

```
type
  MaClasse = class
  private
    Procedure EcrireProp1 ( x : TColor ) ;
    //déclaration privée d'une méthode d'accés
  public
    Property Prop1 : TColor Write EcrireProp1 ;
    // déclaration publique d'une propriété permettant l'accés
    // à la méthode LireProp1 en écriture Seule
  end;
```

La méthode d'accès en écriture **EcririreProp1** doit :

- être une procédure,
- n'avoir qu'un seul paramètre formel passé par constante ou par valeur (**pas par référence**),
- ce paramètre doit être du même type que celui de la propriété.

4.4 Surcharge de propriétés

Surcharge permettant d'augmenter la visibilité

Lorsqu'une propriété est déclarée dans une classe, on peut la surcharger dans les classes dérivées en augmentant son niveau de visibilité.

MaClasse = **class** **private** Fchamp : integer ; **protected** **property** propr1 : integer **read** Fchamp **write** Fchamp ; **end;**	MaFille = **class** (MaClasse) **private** Fchamp : integer ; **public** **property** propr1 : integer **read** Fchamp **write** Fchamp ; end;

Surcharge permettant de redéfinir un spécificateur existant

On ne peut pas supprimer de spécificateur.
Par contre on peut modifier le/les spécificateur(s) existant(s) ou ajouter le spécificateur manquant.

MaClasse = **class**
  private
  Fchamp : integer ;
 	
  public
  property propr1 : integer **read** Fchamp ;
  property propr2 : integer **read** Fchamp ;
end;
	MaFille = **class** (MaClasse)
	  private
	  Fchamp : integer ;
	  function lirePropr2 : integer ;
	  public
	  property propr1 : integer **read** Fchamp **write** Fchamp ;
	  property propr2 : integer **read** lirePropr2 ;
	
	end;

Redéfinition et masquage d'une propriété

Une propriété redéfinie dans une classe remplace l'ancienne avec ses nouveaux attributs, son nouveau type.

MaClasse = **class**
  private
  Fchamp : integer ;
  protected
  property propr1 : integer **read** Fchamp
  write Fchamp ;
end;
	MaFille = **class** (MaClasse)
	  private
	  FNom : string ;
	  public
	  property propr1 : string **read** Fnom ;
	
	end;

Dés qu'une redéclaration de propriété contient une redéfinition de type, elle masque automatiquement la propriété parent héritée et la remplace entièrement. Elle doit donc être redéfinie avec au moins un spécificateur d'accès.

5.3 Polymorphisme avec Delphi

Plan de ce chapitre:

Polymorphisme d'objet

- Introduction
- Instanciation et utilisation dans le même type
- Instanciation et utilisation dans un type différent
- Polymorphisme implicite
- Instanciation dans un type descendant
- Polymorphisme explicite par transtypage
- Utilisation pratique du polymorphisme d'objet
- instanciation dans un type ascendant

Polymorphisme de méthode

- Introduction
- Vocabulaire et concepts
- Surcharge dans la même classe
- Surcharge dans une classe dérivée
- Surcharge dynamique dans une classe dérivée
- Répartition des méthodes en Delphi
- Réutilisation de méthodes avec inherited

Polymorphisme de classe abstraite

- Introduction
- Vocabulaire et concepts

Exercice traité sur le polymorphisme

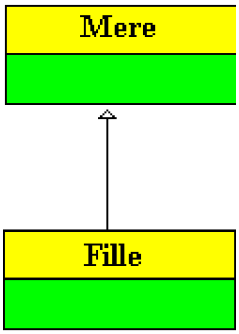
1. Polymorphisme d'objet

Conversion de références d'objet entre classe et classe dérivée

Il existe un concept essentiel en POO désignant la capacité d'une hiérarchie de classes à fournir différentes implémentations de méthodes portant le même nom et par corollaire la capacité qu'ont des objets enfants de modifier les comportements hérités de leur parents. Ce concept d'adaptation à différentes "situations" se dénomme le **polymorphisme** qui peut être implémenté de différentes manières.

Polymorphisme d'objet - *définition générale*
C'est une interchangeabilité entre variables d'objets de classes de la même hiérarchie sous certaines conditions, que dénommons le polymorphisme d'objet.

Soit une classe **Mere** et une **Fille** héritant de la classe **Mere** :



Les objets peuvent avoir des comportements polymorphes (s'adapter et se comporter différemment selon leur utilisation) licites et des comportements polymorphes dangereux selon les langages.

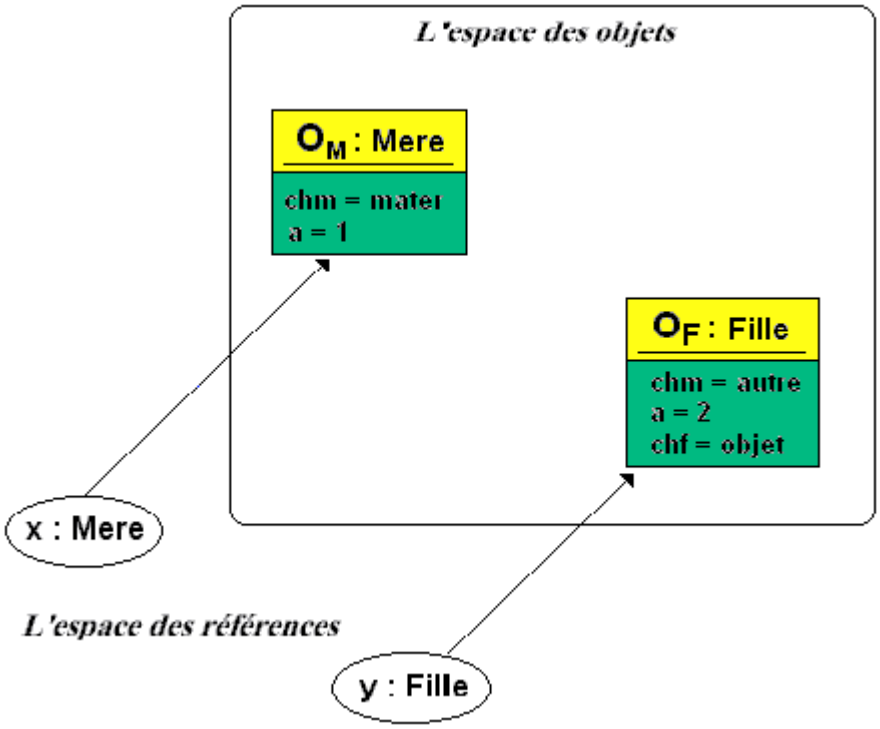
Dans un langage dont le modèle objet est la référence (un objet est un couple : référence, bloc mémoire) comme **C++, C#, Delphi ou Java**, il y a découplage entre les actions statiques du compilateur et les actions dynamiques du système d'exécution selon le langage utilisé le compilateur protège ou non statiquement des actions dynamiques sur les objets une fois créés. C'est la déclaration et l'utilisation des variables de références qui autorise ou non les actions licites grâce à la compilation.

Supposons que nous ayons déclaré deux variables de référence, l'une de classe **Mere**, l'autre de classe **Fille**, une question qui se pose est la suivante : au cours du programme quel genre d'affectation et d'instanciation est-on autorisé à effectuer sur chacune de ces variables. L'héritage permet une variabilité entre variables d'objets de classes de la même hiérarchie, c'est cette variabilité que dénommons le **polymorphisme d'objet**.

Nous allons dès lors envisager toutes les situations possibles et les évaluer, les exemples appuyant le texte sont présentés en Delphi.

instanciation dans le type initial et utilisation dans le même type

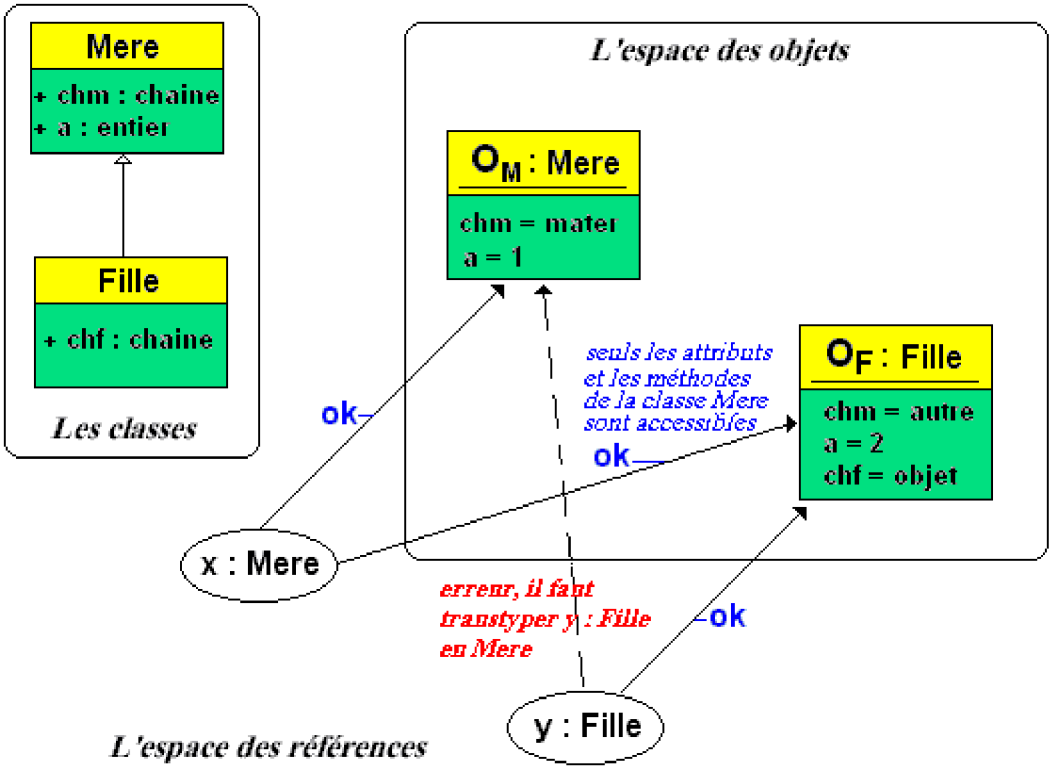
Il s'agit ici d'une utilisation la plus classique qui soit, dans laquelle une variable est utilisée dans son type de définition initial.




```
var
   x,u : Mere;
   y,v : Fille ;
.....
x : = Mere.Create ; // instanciation dans le type initial
u := x; // affectation de références du même type
y : = Fille.Create ;  // instanciation dans le type initial
v := y; // affectation de références du même type
```

instanciation dans le type initial et utilisation différente

Il s'agit ici de l'utilisation licite commune à tous les langages cités plus haut, nous illustrons le discours en explicitant deux champs de la classe Mere (chm:chaine et a:entier) et un champ supplémentaire (chf:chaine) dans la classe Fille. il existe 3 possibilités différentes, la figure ci-dessous indique les affectations possibles :




```
var
  x , ObjM : Mere;
  y , ObjF : Fille;
```

Delphi

```
ObjM := Mere.Create; // instanciation dans le type initial
ObjF := Fille.Create; // instanciation dans le type initial
 x := ObjM; // affectation de références du même type
 x := ObjF; // affectation de références du type descendant implicite
 y := ObjF; // affectation de références du même type
 y := Fille(ObjM); // affectation de références du type ascendant explicite mais dangereux si ObjM est uniquement Mere
```

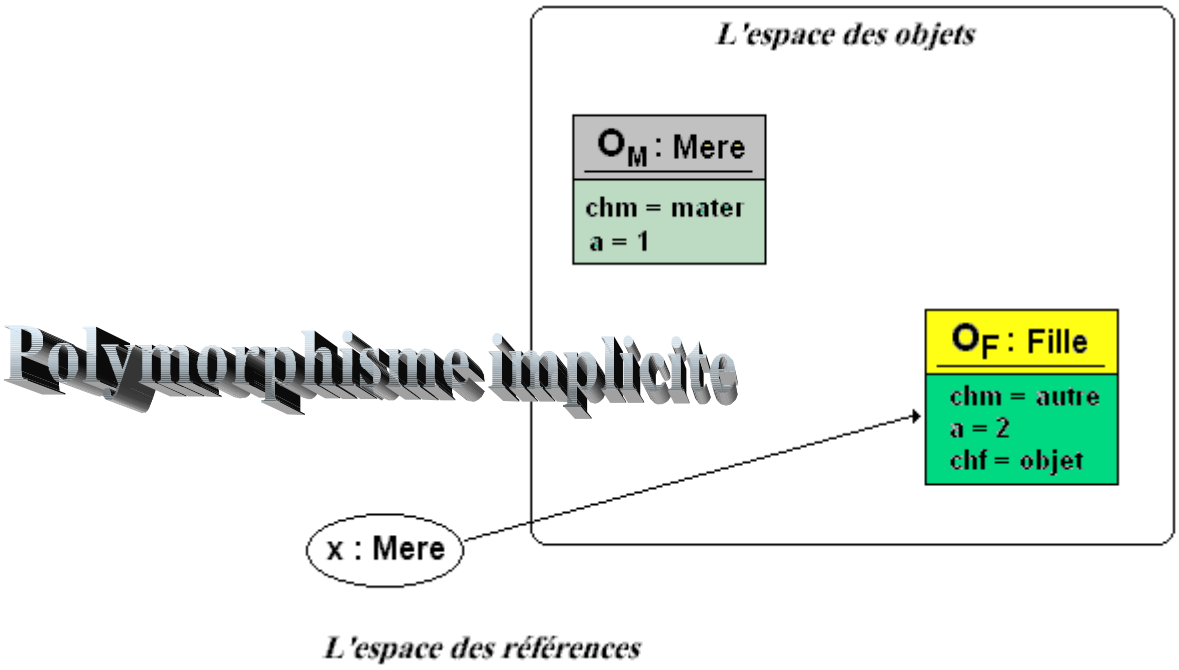
Les trois possibilités sont :

- L'instanciation et l'utilisation de références dans le même type
- L'affectation de références : polymorphisme implicite
- L'affectation de références : polymorphisme par transtypage d'objet

La dernière de ces possibilités pose un problème d'exécution lorsqu'elle est mal employée !

Polymorphisme d'objet implicite

Dans l'exemple précédent le compilateur accepte le transtypage 'y :=Fille(ObjM)' car il autorise un polymorphisme d'objet de classe ascendante vers une classe descendante (c'est à dire que ObjM peut se référer implicitement à tout objet de classe **Mere** ou de toute classe **descendante** de la classe Mere).



Polymorphisme implicite

Nous pouvons en effet dire que **x** peut se référer implicitement à tout objet de classe **Mere** ou de **toute classe héritant** de la classe **Mere** :

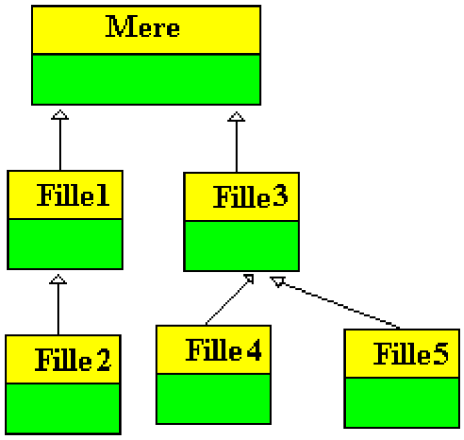


fig - 1

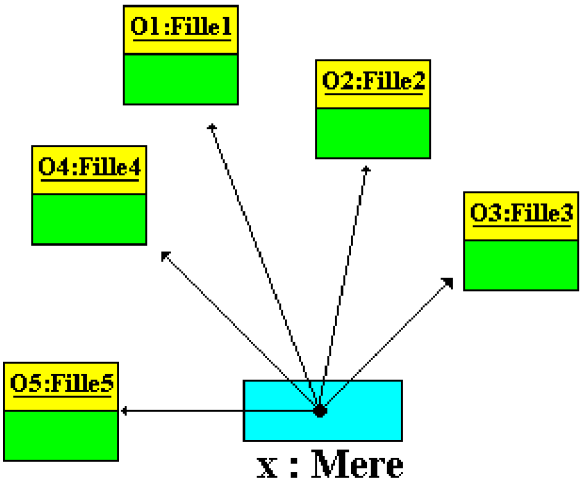


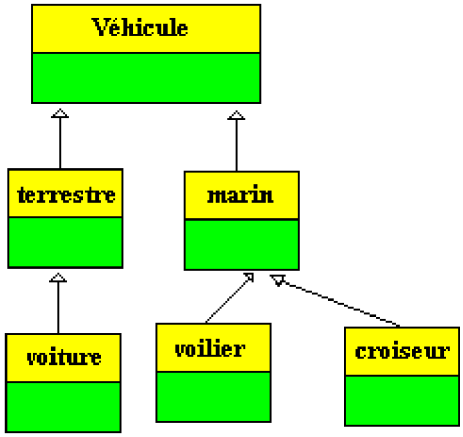
fig - 2

Dans la figure fig-1 ci-dessus, une hiérarchie de classes decendant toutes de la classe **Mere**.

Dans la figure fig-2 ci-dessus le schéma montre une référence de type **Mere** qui peut '**pointer**' vers n'importe quel objet de classe descendante (polymorphisme d'objet).

Exemple pratique tiré du schéma précédent

Le polymorphisme d'objet est typiquement fait pour représenter des situations pratiques figurées ci-dessous : (Mere=vehicule, Fille1=terrestre, Fille2=voiture, Fille3=marin, Fille4=voilier, Fille5=croiseur)



Une hiérarchie de classes de véhicules descendant toutes de la classe mère **Vehicule**.

Déclaration de cette hiérarchie en Delphi

Vehicule = **class** ..... **end;** terrestre = **class** (Vehicule) ..... **end;** voiture = **class** (terrestre) ..... **end;**	Marin = **class** (Vehicule) ..... **end;** voilier = **class** (marin) ..... **end;** croiseur = **class** (marin) ..... **end;**

On peut énoncer le fait qu'un véhicule peut être de plusieurs sortes : soit un **croiseur**, soit une **voiture**, soit un véhicule **terrestre** etc...

En traduisant cette phrase en termes informatiques :

Si l'on déclare une référence de type véhicule **(var** x : **vehicule**) elle pourra pointer vers n'importe quel objet d'une des classe filles de la classe **vehicule**.

Polymorphisme implicite = création d'objet de classe descendante référencé par une variable parent

Quand peut-on écrire x := y sur des objets ?

D'une façon générale vous pourrez toujours écrire des affectations entre deux références d'objets :

var

x : Classe1
y : Classe2
.....
x := y;

Delphi

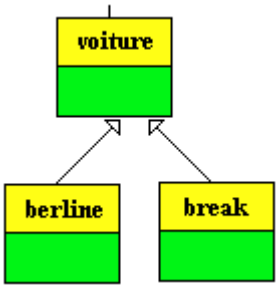
si et seulement si Classe2 est une classe descendante de Classe1.

instanciation dans un type descendant

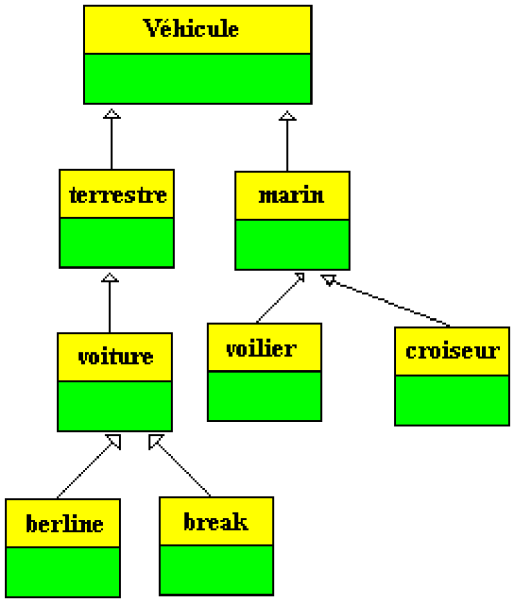
Polymorphisme par création d'objet de classe descendante

Dans ce paragraphe nous signalons qu'il est tout à fait possible, du fait du transtypage implicite, de créer un objet de classe descendante référencé par une variable de classe parent.

Ajoutons 2 classes à la hiérarchie des véhicules :



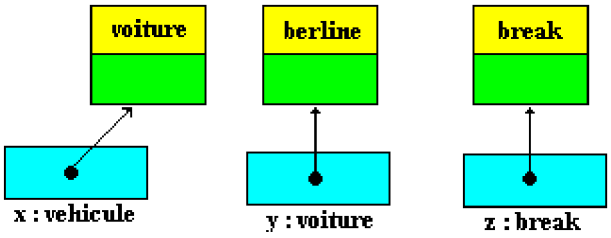
La nouvelle hiérarchie est la suivante :



Ensuite nous déclarons 3 références de type **x:vehicule**, **y:voiture** et **z:break , puis** nous créons 3 objets de classe **voiture**, **berline** et **break**, il est possible de créer directement un objet de classe descendante à partir d'une référence de classe mère :

- on crée une **voiture** référencée par la variable x de classe **vehicule**,
- on crée une **berline** référencée par la variable y de classe **voiture**,
- enfin on crée un **break** référencé par la variable z de classe **break**.

Réécrivons ces phrases afin de comprendre à quel genre de situation pratique cette opération correspond :

on crée un **véhicule** du type **voiture** on crée une **voiture** de type **berline** enfin on crée un **break** de type **break**	

```
var
 x : vehicule;
 y : voiture;
 z : break;
...
x := voiture.Create; // objet de classe enfant voiture référencé par x de classe parent vehicule
y := berline.Create; // objet de classe enfant berline référencé par x de classe parent voiture
z := break.Create; // instanciation dans le type initial
```

Polymorphisme d'objet explicite par transtypage

Reprenons le code précédent en extrayant la partie qui nous intéresse :

```
var
  x , ObjM : Mere;
  y : Fille;

ObjM := Mere.Create; // instanciation dans le type initial
 x := ObjM; // affectation de références du même type
 y := Fille(ObjM); // affectation de références du type ascendant explicite licite
```

Nous avons signalé que l'affectation y := Fille(ObjM) pouvait être dangereuse si ObjM pointe vers un objet purement de type Mere. Voyons ce qu'il en est.

Nous avons vu plus haut qu'une référence de type parent peut '**pointer**' vers n'importe quel objet de classe descendante. Si l'on sait qu'une reférence **x** de classe parent, pointe vers un objet de classe enfant, on peut en toute sureté procéder à une affectation de cette reférence à une autre reférence **y** définie comme reférence classe enfant (opération y := x).

La situation informatique est la suivante :

- on déclare une variable x de type Mere,
- on déclare une variable y de type Fille héritant de Mere,
- on instancie la variable x dans le type descendant Fille (polymorphisme implicite).

Il est alors possible de faire "pointer" la variable y (de type Fille) vers l'objet (de type Fille) auquel se réfère x en effectuant une affectation de références.

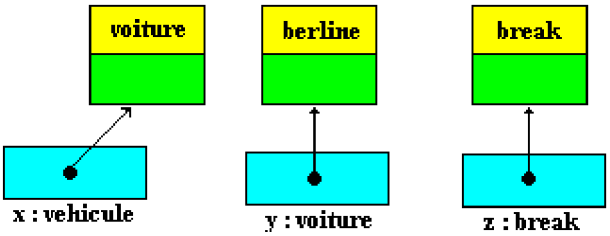
Toutefois le compilateur refusera l'écriture y := x, il suffit de lui indiquer qu'il faut

transtyper la variable de référence x et la considérer dans cette instruction comme une reférence sur un enfant

```
var
 x : Mere;
 y  : Fille;
```

```
x := Mere.Create; // instanciation dans le type initial
y := Fille(ObjM); // affectation de références du type ascendant explicite licite
```

Dans la dernière instruction, la **reférence ObjM est transtypée** en type Fille, de telle manière que le compilateur puisse faire pointer y vers l'objet déjà pointé par ObjM. En reprenant l'exemple pratique de la hiérarchie des véhicules :

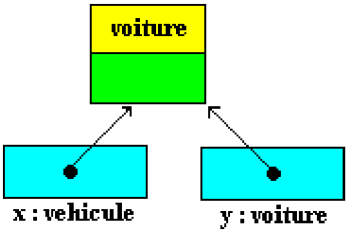


Puisque x pointe vers un objet de type **voiture** toute variable de référence voiture acceptera de pointer vers cet objet, en particulier la variable y :voiture après transtypage de la référence de x.

```
var
 x : vehicule;
 y : voiture;
...
x := voiture.Create; // objet de classe enfant voiture référencé par x de classe parent vehicule
y := voiture ( x ); // transtypage
```

En Delphi l'affectation s'écrit par application de l'opérateur de transtypage :

y := voiture (x);



ATTENTION

- La validité du transtypage n'est pas vérifiée statiquement par le compilateur, donc si votre variable de référence pointe vers un objet qui n'a pas la même nature que l'opérateur de transtypage, c'est de l'exécution qu'il y aura production d'un message d'erreur indiquant le transtypage impossible.
- Il est donc impératif de tester l'appartenance à la bonne classe de l'objet à transtyper avant de le transtyper, les langages C#, Delphi et Java disposent d'un opérateur permettant de tester cette appartenance ou plutôt l'appartenance à une hiérarchie.

L'opérateur "is" en

L'opérateur **is**, qui effectue une vérification de type dynamique, est utilisé pour vérifier quelle est effectivement la classe d'un objet à l'exécution.

L'expression : objet **is** classeT

renvoie True si objet est une instance de la classe désignée par **classeT** ou de l'**un de ses descendants**, et False sinon. Si objet a la valeur nil, le résultat est False.

L'opérateur "as" en Delphi

L'opérateur **as** est un opérateur de transtypage de référence d'objet semblable à l'opérateur ( ).
L'opérateur **as** fournit la valeur **null** en cas d'échec de conversion alors que l'opérateur ( ) lève une exception.

(objet **as** classeT) renvoie une référence de type classeT

Exemple d'utilisation des deux opérateurs :

```
var x : classeT;
if objet is classeT then
  x := objet as classeT ;
```

Utilisation pratique du polymorphisme d'objet

Le polymorphisme d'objet associé au transtypage est très utile dans les paramètres des méthodes.

Lorsque vous déclarez une méthode P avec un paramètre formel de type ClasseT :

```
procedure P( x : ClasseT );
begin
   ........
end;
```

Vous pouvez utiliser lors de l'appel de la procédure P n'importe quel paramètre effectif de ClasseT ou bien d'une quelconque classe descendant de ClasseT et ensuite à l'intérieur de la procédure vous transtypez le paramètre. Cet aspect est abondamment utilisé en Delphi lors de la création de gestionnaires d'événements communs à plusieurs objets :

```
procedure P1( Sender : Tobject );
begin
   if Sender is TEdit then
      TEdit(Sender).text := 'ok'
   else
     if Sender is TButton then
      TButton(Sender).caption := 'oui'
  ............
end;
```

Autre exemple avec une méthode P2 personnelle sur la hiérarchie des véhicules définies plus haut :

```
procedure P2( Sender : vehicule );
begin
    if  Sender is voiture then
      voiture(Sender).  .......
   else
     if Sender is voilier then
      voilier(Sender). .......
  .............
end;
```

2. Polymorphisme de méthode

Introduction

Lorsqu'une classe enfant hérite d'une classe mère, des méthodes supplémentaires nouvelles peuvent être implémentées dans la classe enfant mais aussi des méthodes des parents redéfinies pour obtenir des implémentations différentes.

Une classe dérivée hérite de tous les membres de sa classe parent ; c'est-à-dire que tous les membres du parent sont disponibles pour l'enfant, rappelons qu'une méthode est un membre qualifiant un comportement d'un objet de la classe. En POO on distingue deux catégories de méthodes selon les besoins des applications et du polymorphisme : **les méthodes statiques et les méthodes dynamiques**.

2.1 Vocabulaire et concepts généraux :

- L'action qui consiste à donner le même nom à plusieurs méthodes dans la même classe ou d'une classe parent à une classe enfant, se dénomme d'une manière générale la **surcharge de nom de méthode** (avec ou non la même signature).
- Le vocabulaire n'étant pas stabilisé selon les auteurs (surcharge, redéfinition, substitution,...) nous employerons les mots **redéfinition, surcharge dynamique ou substitution dans le même sens**, en précisant lorsque cela s'avérera nécessaire de quel genre de laison il s'agit.

> Les actions des méthodes héritées du parent peuvent être modifiés par l'enfant de deux manières, selon le type de liaison du code utilisé pour la méthode (**la liaison statique ou précoce** ou bien **la liaison dynamique ou retardée**).

Les deux modes de liaison du code d'une méthode

La liaison statique ou précoce (early-binding) :

- Lorsqu'une méthode à liaison statique est invoquée dans le corps d'un programme, le compilateur établit immédiatement dans le code appelant l'adresse précise et connue du code de la méthode à invoquer. Lors de l'exécution c'est donc toujours le même code invoqué.

La liaison dynamique ou retardée (lazy-binding) :

- Lorsqu'une méthode à liaison dynamique est invoquée dans le corps d'un programme, le compilateur **n**'établit **pas** immédiatement dans le code appelant l'adresse de la méthode à invoquer. Le compilateur met en place un mécanisme de reférence (référence vide lors de la compilation) qui, lors de l'exécution, désignera (pointera vers) le code que l'on voudra invoquer; on pourra donc invoquer des codes différents.

2.2 Surcharge dans la même classe :

Dans une classe donnée, plusieurs méthodes peuvent avoir le même nom, mais les signatures des méthodes ainsi surchargées doivent **obligatoirement** être différentes et peuvent **éventuellement** avoir des niveaux de visibilité différents.

Nous avons déjà vu les bases de ce type de surcharge lors de l'étude de Delphi et la POO. Soit par exemple, la classe ClasseA ci-dessous, ayant 3 méthodes de même nom P, elles sont surchargées dans la classe selon 3 signatures différentes :

```
Classe A
      public methode P(x,y);
      privé methode P(a,b,c);
      protégé methode P( );
finClasse A
```

La première surcharge de P dispose de 2 paramètres, la seconde de 3 paramètres, la dernière enfin n'a pas de paramètres. C'est le compilateur du langage qui devra faire le choix pour sélectionner le code de la bonne méthode à utiliser. Pour indiquer ce genre de surcharge, en Delphi il faut utiliser un qualificateur particulier dénoté **overload**.

Syntaxe de l'exemple en Delphi, en Java et en C# :

Delphi	Java - C#

ClasseA = **class**
public
procedure P(x,y : integer);**overload**;
private
procedure P(a,b,c : string); **overload**;
protected
procedure P;**overload**;
end;
	class ClasseA {
	public void P(**int** x,y){ }
	private void P(**String** a,b,c){ }
	protected void P( ){ }
	}

Utilisation pratique : permettre à une méthode d'accepter plusieurs types de paramètres en conservant le même nom, comme dans le cas d'opérateur arithmétique travaillant sur les entiers, les réels,...

Exemple de code Delphi :

```
ClasseA = class
  public
     procedure P(x,y : integer);overload;
     procedure P(a,b,c : string);overload;
     procedure P;overload;
end;
   var Obj:ClasseA;
   .....
   Obj := ClasseA.create;
   Obj.P( 10, 5 );
   Obj.P( 'abc', 'ef', 'ghi' );
   Obj.P;
```

2.3 Surcharge statique dans une classe dérivée :

D'une manière générale, Delphi et C# disposent **par défaut** de la notion de méthode statique, Java n'en dispose pas sauf dans le cas des méthodes de classes. Dans l'exemple ci-dessous en Delphi et en C#, les trois méthodes P,Q et R sont à liaison statique dans leur déclaration par défaut sans utiliser de qualificateur spécial.

Delphi	C#

ClasseA = **class**
public
procedure P(x,y : integer);
private
procedure Q(a,b,c : string);
protected
procedure R;
end;
	class ClasseA {
	public void P(**int** x,y){ }
	private void Q(**String** a,b,c){ }
	protected void R( ){ }
	}

Une classe dérivée peut **masquer** une méthode à liaison statique héritée en définissant une nouvelle méthode avec **le même nom**.

Si vous déclarez dans une **classe dérivée**, une méthode ayant le même nom qu'une méthode à liaison statique d'une **classe ancêtre**, la nouvelle méthode **remplace** simplement la méthode héritée **dans la classe dérivée**.

Dans ce cas nous employerons aussi le mot de **masquage** qui semble être utilisé par beaucoup d'auteurs pour dénommer ce remplacement, car il correspond bien à l'idée d'un masquage "local" dans la classe fille du code de la méthode de la classe parent par le code de la méthode fille.

Ci-dessous un exemple de hiérarchie de classes et de masquages successifs licites de méthodes à liaison statiques dans certaines classes dérivées avec ou sans modification de visibilité :

Classe A public statique methode P; privé statique methode Q; protégé statique methode R; **finClasse A**	**Classe B hérite de Classe A** public statique methode P; privé statique methode Q; protégé statique methode R; **finClasse B**	**Classe C hérite de Classe B** protégé statique methode P; privé statique methode Q; **finClasse C**
Classe D hérite de Classe C public statique methode P; **finClasse D**	**Classe E hérite de Classe D** protégé statique methode P; **finClasse E**	**Classe F hérite de Classe E** privé statique methode P; public statique methode R; **finClasse F**

Dans le code d'implémentation de la Classe F :

La **méthode P** utilisée est celle qui définie dans la Classe F et elle masque la **méthode P** de la Classe E.
La **méthode Q** utilisée est celle qui définie dans la Classe C.
La **méthode R** utilisée est celle qui définie dans la Classe F et elle masque la **méthode R** de la Classe B

Soit en Delphi l'écriture des classes ClasseA et ClasseB de la hiérarchie ci-haut :

Delphi	Explications
ClasseA = **class** **public** **procedure** P(x,y : integer); **private** **procedure** Q(a,b,c : string); **protected** **procedure** R; **end;**	Dans la classe ClasseB : La méthode **procedure** P(u : char) surcharge statiquement (masque) avec une autre signature, la méthode héritée de sa classe parent **procedure** P(x,y : integer).

ClasseB = **class** (ClasseA) **public** **procedure** P(u : char); **private** **procedure** Q(a,b,c : string); **protected** **procedure** R(x,y : real); **end;**	La méthode **procedure** Q(a,b,c : string) surcharge statiquement (masque) avec la même signature, la méthode héritée de sa classe parent **procedure** Q(a,b,c : string). La méthode **procedure** R(x,y : real) surcharge statiquement (masque) avec une autre signature, la méthode héritée de sa classe parent **procedure** R.

Utilisation pratique : Possibilité notamment de définir un nouveau comportement lié à la classe descendante et éventuellement de changer le niveau de visibilité de la méthode.

Exemple de code Delphi :

ClasseA = **class** **public** **procedure** P(x,y : integer); **procedure** Q(a,b,c : string); **procedure** R; **end;** ClasseB = **class** (ClasseA) **public** **procedure** P(u : char); **procedure** Q(a,b,c : string); **procedure** R(x,y : real); **end;**	..... **var** ObjA:ClasseA; ObjB:ClasseB; ..... ObjA := ClasseA.create; ObjA.P(10, 5); ObjA.Q('abc', 'ef', 'ghi'); ObjA.R; ..... ObjB := ClasseB.create; ObjB.P('g'); ObjB.Q('abc', 'ef', 'ghi'); ObjB.R(1.2, -5.36);

2.4 Surcharge dynamique dans une classe dérivée :

Un type dérivé peut **redéfinir** (**surcharger dynamiquement**) une **méthode à liaison dynamique héritée**. On appelle aussi **virtuelle** une telle méthode à liaison dynamique, nous utiliserons donc souvent ce raccourci de notation pour désigner une méthode surchargeable dynamiquement.

L'action de redéfinition fournit une **nouvelle définition de la méthode** qui sera appelée en fonction du type de l'objet **au moment de l'exécution** et **non** du type de la variable de reférence connue **au moment de la compilation**.

Ci-dessous un exemple de hiérarchie de classes et de redéfinitions (surcharges dynamiques) successives fictives de méthodes à liaison dynamique dans certaines classes dérivées, pour les modifications de visibilité il faut étudier le manuel de chaque langage :

Classe A public dynamique methode P; privé dynamique methode Q; protégé dynamique methode R; **finClasse A**	**Classe B hérite de Classe A** public dynamique methode P; privé dynamique methode Q; protégé dynamique methode R; **finClasse B**	**Classe C hérite de Classe B** protégé dynamique methode P; privé dynamique methode Q; **finClasse C**
Classe D hérite de Classe C public dynamique methode P; **finClasse D**	**Classe E hérite de Classe D** protégé dynamique methode P; **finClasse E**	**Classe F hérite de Classe E** privé dynamique methode P; public dynamique methode R; **finClasse F**

Remarque pratique :

> Une méthode redéfinissant une méthode virtuelle peut selon les langages changer le niveau de visibilité (**il est conseillé de laisser la nouvelle méthode redéfinie au moins aussi visible que la méthode virtuelle parent**).

Tableau comparatif liaison dynamique-statique

Liaison statique	Liaison dynamique
Lors d'un appel pendant l'exécution leur liaison est très rapide car le compilateur a généré l'adresse précise du code de la méthode lors de la compilation.	Lors d'un appel pendant l'exécution leur liaison plus lente car l'adresse précise du code de la méthode est obtenu par un processus de recherche dans une structure de données.
Une telle méthode fonctionne comme une procédure ou fonction d'un langage non orienté objet et ne permet pas le polymorphisme. Car lors d'un appel pendant l'exécution c'est toujours le même code qui est exécuté quel que soit le type de l'objet qui l'invoque.	Une telle méthode autorise le polymorphisme, car bien que portant le même nom dans une hiérarchie de classe, lors d'un appel pendant l'exécution c'est toujours le type de l'objet qui l'invoque qui déclenche le mécanisme de recherche du code adéquat.

2.5 La répartition des méthodes en Delphi

Le terme de **répartition** des méthodes est synonyme de **liaison** et fait référence à la façon dont un programme détermine où il doit rechercher le code d'une méthode lorsqu'il rencontre un appel à cette méthode.

En Delphi, il existe trois modes de répartition des méthodes qui peuvent être :

Statiques ,
Virtuelles,
Dynamiques.

Méthodes statiques en Delphi

Les méthodes statiques de Delphi sont des **méthodes à liaison précoce**.

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;    //statique
 end;

 SousClasse=class(MaClasse)
   procedure Trois;    //statique
 end;
```

Dans SousClas nous avons 3 méthodes statiques :
Un (celle de la mère) , Deux (celle de la mère), Trois (celle de la fille).

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```

Appel de **Un** de MaClasse

Appel de **Deux** de MaClasse

Appel de **Trois** de sousClasse

Voyons ce qui se passe lorsque dans la classe fille on tente de "redéfinir" une méthode héritée de la classe mère. Ci-après nous tentons de "redéfinir" la méthode Deux :

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;   //statique
 end;

 SousClasse=class(MaClasse)
   procedure Deux;   //statique
   procedure Trois;  //statique
 end;
```

Dans SousClas nous avons 3 méthodes statiques :
Un (celle de la mère) , Deux (celle de la mère), Trois (celle de la fille).

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```

Appel de **Un** de MaClasse

Appel de **Deux** de MaClasse

Appel de **Trois** de sousClasse

Lors de l'exécution de l'appel **Obj.Deux**, rien n'a changé. En effet lors de la compilation la variable **Obj** est déclarée de type **MaClasse**, c'est donc l'adresse du code de la méthode **Deux** de la classe **MaClasse** qui est liée. Le type **SousClasse** de l'objet réel vers lequel pointe **Obj** pendant l'exécution n'a aucune influence sur le mode de répartition :

Important

- Cela signifie qu'il est impossible de redéfinir une méthode statique P; **c'est toujours le même code** qui est exécuté, quelque soit la classe dans laquelle P est appelée.
- Si l'on déclare dans une classe dérivée une méthode portant le même nom qu'une méthode statique de la classe mère avec la même signature ou bien avec une signature différente, la nouvelle méthode remplace simplement la méthode héritée dans la classe dérivée, nous dirons qu'elle **masque** la méthode mère.

Masquage avec la même signature

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;   //statique
 end;

 SousClasse=class(MaClasse)
   procedure Deux;  // masque la méthode mère
   procedure Trois;  //statique
 end;
```

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
```

Appel de **Deux** de MaClasse

Masquage avec une signature différente

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux;   //statique
 end;

 SousClasse=class(MaClasse)
   procedure Deux ( x : byte ) ;  // masque la méthode mère
   procedure Trois;  //statique
 end;
```

```
Var Obj : MaClasse ;
Obj := SousClasse.Create ;

Obj.Un ;
Obj.Deux ;
Obj.Trois ;
Obj.Deux ( 49 ) ;
```

Appel de **Deux** de MaClasse

Erreur de compilation ! car Obj est de type MaClasse et la signature **Deux (byte)** n'existe pas dans MaClasse.

Méthodes virtuelles en Delphi

Les méthodes virtuelles utilisent un mécanisme de répartition nécessitant une recherche contrairement aux méthodes statiques. Une méthode virtuelle peut être redéfinie dans les classes descendantes sans masquer ses différentes versions dans les classes ancêtres.

A l'opposé d'une méthode statique l'adresse du code de la méthode virtuelle n'est pas déterminée lors de la compilation, mais seulement lors de l'exécution et en fonction du type de l'objet qui l'appelle.

Les méthodes virtuelles de Delphi sont des **méthodes à liaison tardive**.

Pour déclarer une méthode virtuelle, il faut ajouter le qualificateur **virtual** à la fin de la déclaration de l'en-tête de la méthode :

```
procedure P(x,y : integer); virtual ;
```

Comment se passe la liaison dynamique avec Delphi ?

Delphi implante d'une façon classique le mécanisme de liaison dynamique :

- Lors de la compilation, Delphi rajoute à chaque classe une **T**able des **M**éthodes **V**irtuelles (**TMV**). Cette table contient en principal, pour chaque méthode déclarée avec le qualificateur **virtual** :
  - un pointeur sur le code de la méthode,
  - la taille de l'objet lui-même
- Donc chaque objet possède sa propre **TMV** , et elle est unique.
- La **TMV** d'un objet est créée avec des pointeurs vides lors de la compilation, elle est remplie lors de l'exécution du programme (plus précisément lors de l'instanciation de l'objet) car c'est à l'exécution que l'adresse du code de la méthode est connue et donc stockée dans la **TMV**.
- En fait c'est le constructeur de l'objet lors de son instanciation qui lance le stockage dans la **TMV** des adresses de **toutes** les méthodes virtuelles de l'objet, à la fois les méthodes **héritées** et les méthodes **nouvelles**.

Lorsque l'on construit une nouvelle classe héritant d'une autre classe mère, la nouvelle classe récupère dans sa **TMV** toutes les entrées de la **TMV** de sa classe mère, plus les nouvelles entrées correspondant aux méthodes virtuelles déclarées dans la nouvelle classe. Une **TMV** est donc une structure de données qui grossit au cours de l'héritage de classe et peut être assez volumineuse pour des objets de classes situées en fin de hiérarchie.

Redéfinition de méthode virtuelle avec Delphi

Pour redéfinir une méthode virtuelle dans les classes descendantes sans masquer ses différentes versions dans les classes ancêtres, il faut ajouter à la fin de sa déclaration d'en-tête le qualificateur **override**.

Reprenons l'exemple précédent et tentons de "redéfinir" la méthode **Deux** cette fois en la déclarant virtuelle dans la classe mère et redéfinie dans la classe fille, puis comparons le comportement polymorphique de la méthode **Deux** selon qu'elle est virtuelle ou statique :

type MaClasse=**class** Etat:string; **procedure** Un; *//statique* **procedure** Deux; **virtual** ; *//virtuelle* **end**; SousClasse=**class**(MaClasse) **procedure** Deux; **override**; *//redéfinie* **procedure** Trois; *//statique* **end**;	Dans SousClas nous avons 2 méthodes statiques : Un , Trois et une méthode virtuelle redéfinie : Deux **Var** Obj : MaClasse ; Obj := SousClasse.Create ; Obj.Un ; Obj.Deux ; Obj.Trois ; Appel de **Deux** de sousClasse polymorphe
type MaClasse=**class** Etat:string; **procedure** Un; *//statique* **procedure** Deux; *//statique* **end**; SousClasse=**class**(MaClasse) **procedure** Deux; *//statique* **procedure** Trois; *//statique* **end**;	Dans SousClas nous avons 3 méthodes statiques : Un (celle de la mère) , Deux (celle de la mère), Trois (celle de la fille). **Var** Obj : MaClasse ; Obj := SousClasse.Create ; Obj.Un ; Obj.Deux ; Obj.Trois ; Appel de **Deux** de MaClasse non polymorphe

Méthodes dynamiques en Delphi

Les méthodes dynamiques sont des méthodes virtuelles avec un mécanisme de répartition différent, donc les méthodes dynamiques de Delphi sont des **méthodes à liaison tardive**.

technique

Au lieu d'être stockées dans la **T**able des **M**éthodes **V**irtuelles, les méthodes dynamiques sont ajoutées dans une **structure de données de liste** spécifique pour chaque objet (la liste des méthodes dynamiques).

Seules les adresses des méthodes **nouvelles** ou **redéfinies** d'une classe sont rangées dans sa liste.

La liaison précode d'une méthode dynamique héritée s'effectue en recherchant dans la liste des méthodes dynamiques de chaque ancêtre, en remontant la hiérarchie de l'héritage.

Pour déclarer une méthode dynamique, il faut ajouter le qualificateur **dynamic** à la fin de la déclaration de l'en-tête de la méthode :

```
procedure P(x,y : integer); dynamic ;
```

Remarques de Borland

- **Toute méthode** sans qualification **particulière est considérée comme statique** par défaut.
- Comme les méthodes dynamiques ne disposent pas d'entrées dans la table des méthodes virtuelles de l'objet, elles réduisent la quantité de mémoire occupée par les objets.
- Si une méthode est appelée fréquemment, ou si le temps d'exécution est un paramètre important, il vaut mieux déclarer une méthode virtuelle plutôt que dynamique.

Masquage de méthode virtuelle avec Delphi

Pour masquer une méthode virtuelle dans une de ses classes, il suffit de **ne pas** ajouter à la fin de sa déclaration d'en-tête le qualificateur **override**.

Ceci peut se faire de deux façons soit en masquant par une méthode statique, soit en masquant par une méthode dynamique.

Masquage par une méthode dynamique

Dans le code suivant, la méthode Deux déclarée virtuelle dans la classe mère, est masquée par une méthode Deux ayant la même signature et déclarée elle aussi virtuelle dans la classe fille :

```
type
 MaClasse=class
   Etat:string;
   procedure Un;     //statique
   procedure Deux; virtual;     //virtuelle
 end;
 SousClasse=class(MaClasse)
   procedure Un ;     //statique, masque la méthode ancêtre
   procedure Deux;virtual;    // virtuelle, masque la méthode ancêtre, mais elle-même est redéfinissable
 end;
```

comparons le comportement polymorphique de la méthode Deux selon qu'elle est redéfinie, masquée par une méthode statique ou masquée par une méthode virtuelle :

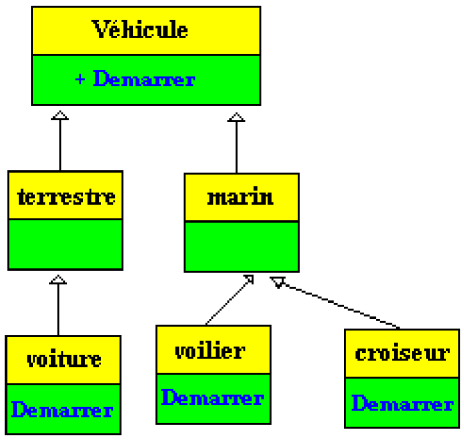
type MaClasse=**class** Etat:string; **procedure** Un; *//statique* **procedure** Deux; **virtual** ; *//virtuelle* **end**; SousClasse=**class**(MaClasse) **procedure** Deux; **override**; *//redéfinie* **procedure** Trois; *//statique* **end**;	Dans SousClas nous avons 2 méthodes statiques : Un , Trois et une méthode virtuelle redéfinie : Deux **Var** Obj : MaClasse ; Obj := SousClasse.Create ; Obj.Un ; Obj.Deux ; Obj.Trois ; Appel de **Deux** de sousClasse polymorphe
type MaClasse=**class** Etat:string; **procedure** Un; *//statique* **procedure** Deux; **virtual** ; *//virtuelle* **end**; SousClasse=**class**(MaClasse) **procedure** Deux; *//masquage* **procedure** Trois; *//statique* **end**;	Dans SousClas nous avons 2 méthodes statiques : Un , Trois et une méthode virtuelle masquée statiquement : Deux **Var** Obj : MaClasse ; Obj := SousClasse.Create ; Obj.Un ; Obj.Deux ; Obj.Trois ; Appel de **Deux** de MaClasse non polymorphe
type MaClasse=**class** Etat:string; **procedure** Un; *//statique* **procedure** Deux; **virtual** ; *//virtuelle* **end**; SousClasse=**class**(MaClasse) **procedure** Deux; **virtual** ; *//masquage* **procedure** Trois; *//statique* **end**;	Dans SousClas nous avons 2 méthodes statiques : Un , Trois et une méthode virtuelle masquée virtuellemnt: Deux **Var** Obj : MaClasse ; Obj := SousClasse.Create ; Obj.Un ; Obj.Deux ; Obj.Trois ; Appel de **Deux** de MaClasse non polymorphe

Nous pouvons conclure de ce tableau de comparaison que le masquage (par une méthode statique ou virtuelle) ne permet jamais le polymorphisme.

Exemple pratique

Le polymorphisme de méthode offre la possibilité de conserver **le même nom de méthode** dans une hiérarchie de classe, afin de ne pas "surcharger" le cerveau de l'utilisateur. Les comportements seront différents selon le type d'objet utilisé.

Par exemple l'action de **Démarrer** dans une hiérarchie de véhicules :



Nous voyons bien que sémantiquement parlant on peut dire qu'une voiture démarre, qu'un voilier démarre, qu'un croiseur démarre, toutefois les actions internes permettant le comportement démarrage ne sont pas les mêmes.

- **Démarrer** dans la classe **voiture** : tourner la clef de contact, engager une vitesse,...
- **Démarrer** dans la classe **voilier** : hisser les voiles, dégager la barre,...
- **Démarrer** dans la classe **croiseur** : lancer les moteurs, modifier la barre,...

L'action Démarrer est polymorphe (car elle s'adapte au type de véhicule qui l'exécute).

Pour traduire en Delphi, ce comportement polymorphe de l'action Démarrer, nous allons utiliser une **méthode virtuelle** que nous **redéfinissons** dans toutes les classes dérivées.

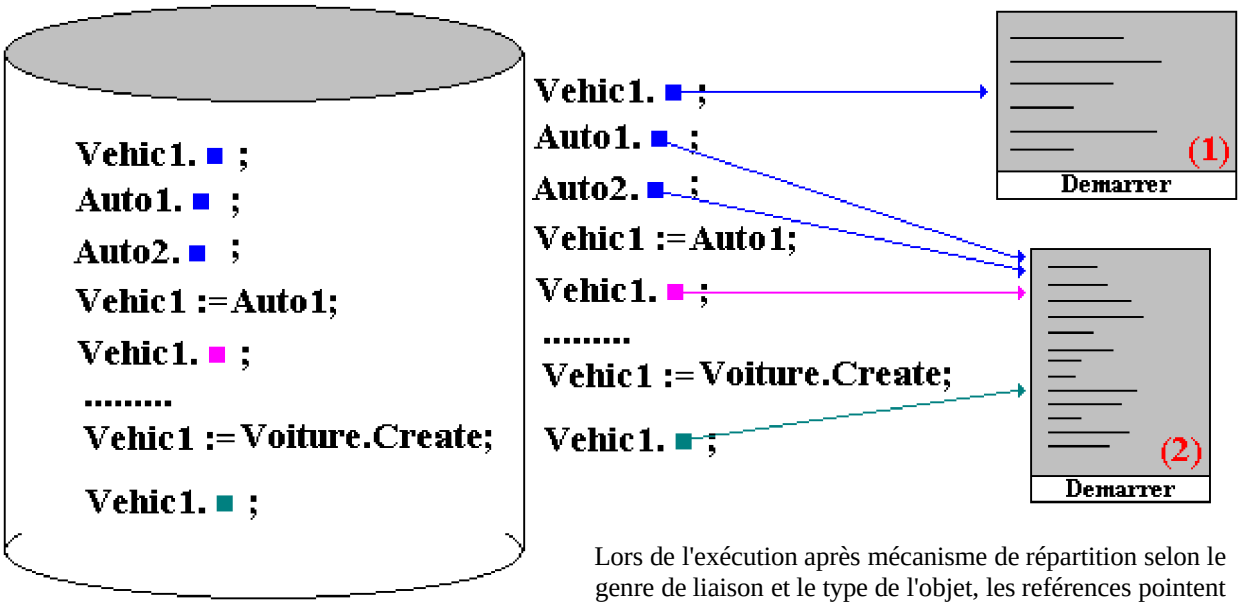
Soit en Delphi l'écriture de l'exemple de la hiérarchie précédente :

Delphi	
Vehicule = **class** **public** **procedure** Demarrer; **virtual**; *//virtuelle* **end;** Terrestre = **class** (Vehicule) ..... **end;** Voiture = **class** (Terrestre) **public** **procedure** Demarrer; **override**; *//redéfinie* **end;**	Marin = **class** (Vehicule) ..... **end;** Voilier = **class** (Marin) **public** **procedure** Demarrer; **override**; *//redéfinie* **end;** Croiseur = **class** (Marin) **public** **procedure** Demarrer; **override**; *//redéfinie* **end;**

Exemple de code d'utilisation :

Vehicule = **class** **public** **procedure** Demarrer; **virtual**; **(1)** **end;** Voiture = **class** (Terrestre) **public** **procedure** Demarrer; **override**; **(2)** **end;** ..... **var** Vehic1 : Vehicule; Auto1, Auto2 : Voiture;	..... Vehic1 := Vehicule.Create; *//instanciation dans le type* Auto1 := Voiture.Create; *//instanciation dans le type* Auto2 := Voiture.Create; *//instanciation dans le type* Vehic1.Demarrer; *//méthode du type Vehicule* **(1)** Auto1.Demarrer; *//méthode du type Voiture* **(2)** Auto2.Demarrer; *//méthode du type Voiture* **(2)** Vehic1 := Auto1; *//pointe vers un objet de type hérité* Vehic1.Demarrer; *// polymorphisme à l'œuvre ici:* **(2)** ..... Vehic1 := Voiture.Create; *//instanciation un type hérité* Vehic1.Demarrer; *// polymorphisme à l'œuvre ici:* **(2)**

Illustrons l'exemple précédent avec une image de la partie de code généré sur la méthode Demarrer et ensuite l'exécution de ce code sur des objets effectifs. Nous avons numéroté **(1)** et **(2)** les deux codes d'implantation de la méthode Demarrer dans la classe parent et dans la classe enfant :



Le code engendré contient des références vides

Lors de l'exécution après mécanisme de répartition selon le genre de liaison et le type de l'objet, les références pointent vers le code de la méthode du type effectif de l'objet.

2.6 Réutilisation de méthodes avec inherited

Le mot réservé **inherited** joue dans Delphi un rôle particulier dans l'implémentation de comportements polymorphiques (il joue un rôle semblable à **super** dans java et **base** dans C#).

Il ne peut qu'apparaître dans une définition de méthode avec ou sans identificateur à la suite.

- Si **inherited** présent dans une methode P de la classeA est suivi par le nom d'une méthode Q, il représente un appel normal de la méthode Q, sauf que la recherche de la méthode à invoquer commence dans l'ancêtre immédiat de la classe de la méthode et remonte la hiérarchie.

- Si **inherited** présent dans une methode P de la classeA, n'est suivi d'aucun nom de méthode, il représente alors un appel à une méthode de même nom P, la recherche de la méthode à invoquer commence dans l'ancêtre immédiat de la classe de la méthode P actuelle et remonte la hiérarchie.

```
procedure classeA.P(x,y : integer);
begin
 inherited ;

end;
```

<=> **inherited** P(x,y); pui recherche et appel de P(x,y) dans les ancêtres de classeA

```
procedure classeA.P(x,y : integer);
begin
  inherited   Q ( x );

end;
```

Recherche et appel de Q (x) dans les ancêtres de classeA

3. Polymorphisme de classes abstraites

Introduction

Les classes abstraites permettent de créer des classes génériques **expliquant certains comportements sans les implémenter** et fournissant une implémentation commune de certains autres comportements pour l'héritage de classes. Les classes abstraites sont un outil précieux pour le **polymorphisme**.

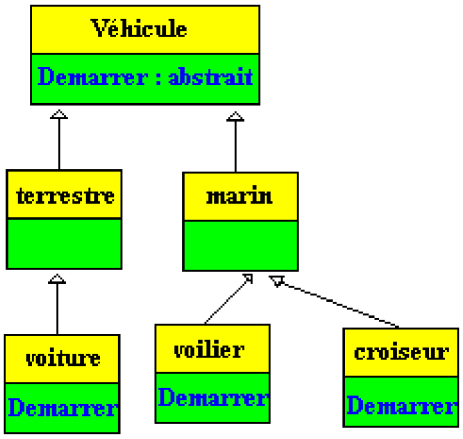
Vocabulaire et concepts :

- Une classe abstraite est une classe qui **ne** peut **pas** être instanciée.
- Une classe abstraite peut contenir des méthodes déjà implémentées.
- Une classe abstraite peut contenir des méthodes **non** implémentées.
- Une classe abstraite est héritable.
- On peut contsruire une hiérarchie de classes abstraites.
- Pour pouvoir construire un objet à partir d'une classe abstraite, il faut dériver une classe non abstraite en une classe implémentant **toutes** les méthodes **non** implémentées.

- Une méthode déclarée dans une classe, **non implémentée** dans cette classe, mais juste définie par la déclaration de sa signature, est dénommée **méthode abstraite**.

- Une méthode abstraite est une méthode à liaison dynamique n'ayant pas d'implémentation dans la classe où elle est déclarée. L' implémentation d'une méthode abstraite est déléguée à une classe dérivée.

Si vous voulez utiliser la notion de classe abstraite pour fournir un comportement polymorphe à un groupe de classes, elles doivent toutes hériter de la même classe abstaite, comme dans l'exemple ci-dessous :

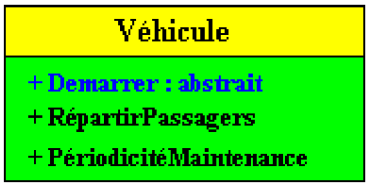


La classe **Véhicule** est abstraite, car la méthode **Démarrer** est abstraite et sert de "modèle" aux futures classes dérivant de **Véhicule**, c'est dans les classes **voiture**, **voilier** et **croiseur** que l'on implémente le comportement précis du genre de démarrage.

Notons au passage que dans la hiérarchie précédente, les classes vehicule **Terrestre** et **Marin** héritent de la classe **Véhicule**, mais n'implémentent pas la méthode abstraite **Démarrer**, ce sont donc par construction des classes abstraites elles aussi.

Les classes abstraites peuvent également **contenir des membres déjà implémentés**. Dans cette éventualité, une classe abstraite propose un certain nombre de **fonctionnalités identiques** pour tous ses futurs descendants (*ceci n'est pas possible avec une interface*).

Exemple : la classe abstraite **Véhicule** n'implémente pas la méthode abstraite **Démarrer**, mais fournit et implante une méthode "**RépartirPassagers**" de répartition des passagers à bord du véhicule (fonction de la forme, du nombre de places, du personnel chargé de s'occuper de faire fonctionner le véhicule...), elle fournit aussi et implante une méthode "**PériodicitéMaintenance**" renvoyant la périodicité de la maintenance obligatoire du véhicule (fonction du nombre de km ou miles parcourus, du nombre d'heures d'activités,...)



Ce qui signifie que toutes les classes **voiture**, **voilier** et **croiseur** savent comment répartir leurs éventuels passagers et quand effectuer une maintenance, chacune d'elle implémente son propre comportement de démarrage.

Syntaxe de l'exemple en Delphi et en Java :

Delphi	Java

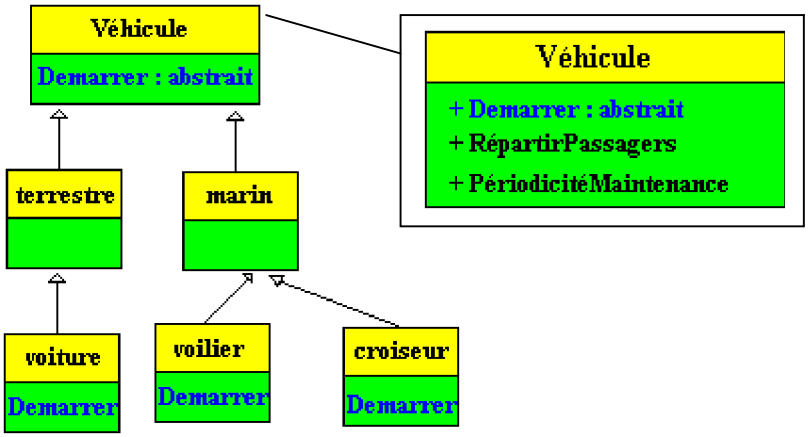
Vehicule = **class**
public
procedure Demarrer; **virtual**;**abstract**;
procedure RépartirPassagers; **virtual**;
procedure PériodicitéMaintenance; **virtual**;
end;
	abstract class ClasseA {
	public abstract void Demarrer( );
	public void RépartirPassagers( );
	public void PériodicitéMaintenance( );
	}

Utilisation pratique des classes abstraites

Utilisez une classe abstraite lorsque vous voulez :

- **regrouper** un ensemble de méthodes présentant des **fonctionnalités identiques**,
- **déléguer** l'implémentation de certaines méthodes à une classe dérivée,
- **disposer** immédiatement de fonctionnalités concrètes pour d'autres méthodes,
- **assurer un comportement polymorphe** aux méthodes dont on délègue l'implantation du code à des classes dérivées.

Exemple de code Delphi pour la hiérarchie ci-dessous :



Soit en Delphi l'écriture d'un exemple tiré de cette hiérarchie :

Delphi

```
Unit UclassVehicules;
    interface
    Vehicule = class
      public
          procedure Demarrer; virtual;abstract;
          procedure RépartirPassagers; virtual;
          procedure PériodicitéMaintenance; virtual;
    end;
    Terrestre = class ( Vehicule )
      public
          procedure PériodicitéMaintenance; override;
    end;
    Voiture = class ( Terrestre )
      public
          procedure Demarrer; override;
          procedure RépartirPassagers; override;
    end;
    Marin = class ( Vehicule )
      public
          procedure PériodicitéMaintenance; override;
    end;
    Voilier = class ( Marin )
      public
          procedure Demarrer; override;
          procedure RépartirPassagers; override;
    end;
    Croiseur = class ( Marin )
      public
          procedure Demarrer; override;
          procedure RépartirPassagers; override;
    end;
    implementation
    //--- les méthodes implantées de la classe abstraite Vehicule :
     procedure Vehicule.RépartirPassagers;
     begin
       ..........
     end;
     procedure Vehicule.PériodicitéMaintenance;
     begin
```

```
      ..........
    end;
   //--- les méthodes implantées de la classe abstraite Terrestre :
    procedure Terrestre.PériodicitéMaintenance;
    begin
      ..........
    end;
   //--- les méthodes implantées de la classe abstraite Marin :
    procedure Marin.PériodicitéMaintenance;
    begin
      ..........
    end;
   //--- les méthodes implantées de la classe Voiture :
    procedure Voiture.Demarrer;
    begin
      ..........
    end;
    procedure Voiture.RépartirPassagers;
    begin
      ..........
    end;
   //--- les méthodes implantées de la classe Voilier :
    procedure Voilier.Demarrer;
    begin
      ..........
    end;
    procedure Voilier.RépartirPassagers;
    begin
      ..........
    end;
   //--- les méthodes implantées de la classe Croiseur :
    procedure Croiseur.Demarrer;
    begin
      ..........
    end;
    procedure Croiseur.RépartirPassagers;
    begin
      ..........
    end;
end.
```

Dans cet exemple :

Les classes **Vehicule**, **Marin** et **Terrestre** sont abstraites car aucune n'implémente la méthode abstraite **Demarrer**

Les classes **Marin** et **Terrestre** contiennent chacune une surcharge dynamique implémentée de la méthode virtuelle PériodicitéMaintenance qui est déjà implémentée dans la classe Véhicule.

Les classes **Voiture**, **Voilier** et **Croiseur** ne sont pas abstraites car elles implémentent les (la) méthodes abstraites de leurs parents.

Exercice traité sur le polymorphisme

Objectifs : Comparer le polymorphisme et l'utilisation de tests de type if...then avec l'opérateur is.

On souhaite construire une application utilisant des classes de gestion de structures de données de noms (chaînes). Ces classes devraient être capables de lancer une intialisation de la structure, de trier par ordre croissant les données, de les afficher dans un éditeur de texte. Nous voulons disposer d'une quatrième classe possédant une méthode générale d'édition d'une structure de données après son ordonnancement.

- Au départ nous travaillons sur une classe contenant un tableau, une classe contenant une liste chaînée et une classe contenant un fichier. Chaque classe contient trois méthodes : Tri, Initialiser, et Ecrire qui ne seront qu'esquissées, car c'est la conception de la hiérarchie qui nous intéresse dans cet exemple et non le code interne. Le lecteur peut s'il le souhaite développer un code personnel pour toutes ces méthodes.

LES TYPES DE STRUCTURES DE BASE PROPOSES

Nous proposons de travailler avec les types de données suivants :

type

Element=**record** clef : integer; info : ShortString; **end**;	tableau=**Array**[1..100]**of** Element;
fichier = **file of** Element;	pointeur = ^chainon; chainon=**record** data:Element; suivant:pointeur **end**;

Nous supposons avoir fourni ces informations à deux équipes de développement leur laissant le choix de l'organisation des classes.

- La **première équipe** décide d'implanter les 3 classes à partir de la classe racine (TObject en Delphi).
- La **seconde équipe** de développement a choisi pour le même problème d'implémenter les 3 classes à partir d'une hiérarchie de classes fondée sur une classe abstraite.

CLASSES DESCENDANT DE TObject

Equipe1-1°) L'équipe implémente une gestion de structure de données à partir de classe héritant toutes de la classe racine TObject avec des **méthodes à liaison statique**.

Ttable=**class** T:Tableau; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;	Tliste=**class** L:pointeur; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;	Tfichier=**class** F:fichier; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;

Ce choix permet aux membres de l'équipe n°1 de déclarer et d'instancier des objets dans chaque classe :

var
S1:Ttable; S2:Tliste; S3:Tfichier;

S1:=Ttable.Create; S1.Initialiser; S1.Tri; S1.Ecrire(Form1.Edit1);	S2:=Tliste.Create; S2.Initialiser; S2.Tri; S2.Ecrire(Form1.Edit1);	S3:=Tfichier.Create; S3.Initialiser; S3.Tri; S3.Ecrire(Form1.Edit1);

L'équipe n°1 pourra utiliser le **polymorphisme d'objet** en déclarant une reférence d'objet de **classe racine** puis en l'instanciant selon les besoins dans l'une des trois classes. Il sera alors nécessaire de transtyper la reférence en testant auparavant par sécurité, l'appartenance de l'objet referencé à la bonne classe :

var
S1:TObject;

if S1 **is** Ttable **then**
begin
Ttable(S1).Initialiser;
Ttable(S1).Tri;
Ttable(S1).Ecrire(Form1.Edit1);
End
	if S1 **is** Tliste **then**
	begin
	Tliste(S1).Initialiser;
	Tliste(S1).Tri;
	Tliste(S1).Ecrire(Form1.Edit1);
	end
		if S1 **is** Tfichier **then**
		begin
		Tfichier(S1).Initialiser;
		Tfichier(S1).Tri;
		Tfichier(S1).Ecrire(Form1.Edit1);
		end

<< S1:=Ttable.Create; >>	<< S1:=Tliste.Create; >>	<< S1:=TFichier.Create; >>

Dans l'application, l'équipe n°1 construit une classe **TUseData** qui possède une méthode générale d'édition d'une structure de données après ordonnancement , cette méthode utilise le polymorphisme d'objet pour s'adapter à la structure de données à éditer :

TUseData=**class**
Procedure Editer (x : Tobject);
end ;

C'est le paramètre formel de classe générale TObject qui est polymorphe, les valeurs effectives qu'il peut prendre sont : n'importe quel objet de classe héritant de Tobject.

Code de la méthode Editer de TuseData :

```
Procedure TUseData .Editer ( x : Tobject );
Begin
        if  x  is Ttable then
         begin
           Ttable(x).Tri;
           Ttable(x).Ecrire(Form1.Edit1);
         end
Else
        if  x  is Tliste then
         begin
           Tliste(x).Tri;
           Tliste(x).Ecrire(Form1.Edit1);
         end
Else
        if  x  is Tfichier then
         begin
           Tfichier(x)Tri;
           Tfichier(x).Ecrire(Form1.Edit1);
         end
End;
```

Equipe1-2°) L'équipe livre au client l'application contenant en de multiples endroits un appel à la méthode Editer.

Equipe1-3°) Deux mois après la livraison et la mise en place, le client souhaite rajouter une nouvelle structure de données que nous nommons TDatas (*par exemple de structure d'arbre binaire*).

L'équipe propose de poursuivre la même organisation en créant et en implantant une nouvelle classe dérivant de TObject :	TAutre=**class** Data : TDatas; **procedure** Tri; **procedure** Initialiser; **procedure** Ecrire(Ed:Tedit); **end** ;

Equipe1-4°) L'équipe 1 doit alors reprendre et modifier le code de la méthode **Procedure** Editer (x : Tobject); de la classe TUseData, en y ajoutant le test du nouveau type TDatas comme suit :

```
Procedure TUseData .Editer ( x : Tobject);
Begin
        if  x  is Ttable then …else
         if  x  is Tliste then … else
          if  x  is Tfichier then … else
           if  x  is TDatas then
            begin
              TDatas (x).Tri;
              TDatas (x).Ecrire(Form1.Edit1);
            end;
End;
```

Cette démarche de rajout et de recompilation, les membres de l'équipe n°1 devront l'accomplir dans chaque partie de l'application qui utilise la méthode Editer.

Equipe1-5°) L'équipe n°1 envoie ensuite au client :

- La nouvelle classe et son code,
- ainsi que la nouvelle version de toutes les parties de l'application qui font appel à la méthode Editer dont la précédente version est maintenant caduque.

CLASSES DESCENDANT DE LA MEME CLASSE ABSTRAITE

Equipe2-1°) L'équipe de développement a choisi pour le même problème d'implémenter une hiérarchie de classes fondée sur une classe **abstraite** qui a été nommée TStructData.

```
TStructData=class
      procedure Tri;virtual;abstract;
      procedure Initialiser;virtual;abstract;
      procedure Ecrire(Ed:Tedit);virtual;abstract;
end ;
```

Toutes les autres classes dériveront de TStructData, et possèderont des méthodes à liaisons dynamiques afin d'utiliser ici le polymorphisme de méthodes :

Ttable=**class** (TStructData) T:Tableau; **procedure** Tri;**override**; **procedure** Initialiser;**override**; **procedure** Ecrire(Ed:Tedit);**override**; **end** ;	Tliste=**class** (TStructData) L:pointeur; **procedure** Tri;**override**; **procedure** Initialiser;**override**; **procedure** Ecrire(Ed:Tedit);**override**; **end** ;
Tfichier=**class** (TStructData) F:fichier; **procedure** Tri;**override**; **procedure** Initialiser;**override**; **procedure** Ecrire(Ed:Tedit);**override**; **end** ;	

Ce choix permet aux membres de l'équipe n°2, comme pour ceux de l'équipe n°1, de déclarer et d'instancier des objets dans chaque classe :

var
S1:Ttable; S2:Tliste; S3:Tfichier;

S1:=Ttable.Create; S1.Initialiser; S1.Tri; S1.Ecrire(Form1.Edit1);	S2:=Tliste.Create; S2.Initialiser; S2.Tri; S2.Ecrire(Form1.Edit1);	S3:=Tfichier.Create; S3.Initialiser; S3.Tri; S3.Ecrire(Form1.Edit1);

L'équipe n°2 pourra utiliser le polymorphisme de méthode en déclarant une reférence d'objet de **classe racine abstraite** puis en l'instanciant selon les besoins dans l'une des trois classes. Mais ici, il **ne** sera **pas** nécessaire de transtyper la reférence **ni** de tester auparavant par sécurité, l'appartenance de l'objet référencé à la bonne classe. En effet les méthodes Initialiser, Tri et Ecrire étant virtuelles, c'est le type de l'objet lors de l'exécution qui déterminera le bon choix :

Var S1 : TStrucData;

<< S1:=Ttable.Create; >> << S1:=Tliste.Create; >> << S1:=TFichier.Create; >>
S1.Initialiser; S1.Tri; S1.Ecrire (Form1.Edit1);

Dans l'application, l'équipe n°2 comme l'équipe n°1, construit une classe **TUseData** qui possède une méthode générale d'édition d'une structure de données après ordonnancement, cette méthode utilise le **polymorphisme d'objet** et le **polymorphisme de méthode** pour s'adapter à la structure de données à éditer :

```
TUseData=class
   Procedure Editer ( x : TstructData );
end ;
```

Méthode Editer de TuseData :

Méthode Editer de TuseData : **équipe n°2**	Méthode Editer de TuseData : **équipe n°1**

Procedure TuseData.Editer (x : TstructData);
Begin
x.Tri;
x.Ecrire(Form1.Edit1);
End;
	Procedure TUseData .Editer (x : Tobject);
	Begin
	if x **is** Ttable **then**
	begin
	Ttable(x).Tri;
	Ttable(x).Ecrire(Form1.Edit1);
	end else
	if x **is** Tliste **then**
	begin
	Tliste(x).Tri;
	Tliste(x).Ecrire(Form1.Edit1);
	end else
	if x **is** Tfichier **then**
	begin
	Tfichier(x)Tri;
	Tfichier(x).Ecrire(Form1.Edit1);
	end
	End;

Equipe2-2°) L'équipe livre au client l'application contenant en de multiples endroits un appel à notre méthode Editer.

Equipe2-3°) Deux mois après l'équipe n°2 s'est vu demander comme pour l'autre équipe,

de rajouter une nouvelle structure de données (par exemple de structure d'arbre binaire). L'équipe n°2 crée et implante une nouvelle classe dérivant de la classe abstraite TStructData comme pour les 3 autres classes déjà existantes :

```
TAutre=class (TStructData)
   Data : TDatas;
   procedure Tri;override;
   procedure Initialiser;override;
   procedure Ecrire(Ed:Tedit);override;
end ;
```

Grâce au polymorphisme d'objet et de méthode à liaison dynamique la méthode Editer fonctionne avec toutes les descendants de TstructData.

Equipe2-4°) L'équipe n°2 n'a pas à modifier le code de la méthode **Procedure** Editer (x : TStructData).

Equipe2-5°) Il suffit à l'équipe n°2 d'envoyer au client la nouvelle classe et son code (toutes les parties de l'application qui font appel à la méthode Editer fonctionneront correctement automatiquement) !

Squelettes des méthodes

Les deux équipes ont coopéré entre elles, le code des méthodes est le même quelque soit le choix effectué (hériter de TObject ou hériter d'une classe abstraite).

```
//////////////////// Les tableaux ////////////////////////
 procedure Ttable.Tri;
 begin
  //algorithme de tri d'un tableau
  T[1].info:=T[1].info+' : tableau trié'
 end;

 procedure Ttable.Initialiser;
 begin
  T[1].clef:=100;
  T[1].info:='Durand';//etc...
 end;

 procedure Ttable.Ecrire(Ed:Tedit);
 begin
  Ed.Text:='Clef= '+inttostr(T[1].clef)+'//'+T[1].info
 end;

//////////////////// Les listes chaînées ////////////////////////
 procedure Tliste.Tri;
 begin
  //algorithme de tri d'une liste ...
  L.data.info:=L.data.info+' : liste triée'
 end;

 procedure Tliste.Initialiser;
 begin
```

```
 new(L);
 L.data.clef:=100;
 L.data.info:='Durand';
 L.suivant:=nil  //etc...
end;

procedure Tliste.Ecrire(Ed:Tedit);
begin
 Ed.Text:='Clef= '+inttostr(L.data.clef)+'//'+L.data.info
end;

//////////////////// Les fichiers ////////////////////////
procedure Tfichier.Tri;
var UnElement:Element;
begin
 //algorithme de tri d'un fichier ...
 AssignFile(F,'FichLocal');
 reset(F);
 read(F,UnElement);
 CloseFile(F);
 UnElement.info:=UnElement.info+' : fichier trié';
 reset(F);
 write(F,UnElement);
 CloseFile(F)
end;

procedure Tfichier.Initialiser;
var UnElement:Element;
begin
 AssignFile(F,'FichLocal');
 rewrite(F);
 UnElement.clef:=100;
 UnElement.info:='Durand';
 write(F,UnElement); //etc...
 CloseFile(F)
end;

procedure Tfichier.Ecrire(Ed:Tedit);
var UnElement:Element;
begin
 AssignFile(F,'FichLocal');
 reset(F);
 read(F,UnElement);
 Ed.Text:='Clef= '+inttostr(UnElement.clef)+'//'+UnElement.info;
 CloseFile(F);
end;

end.
```

Conclusion

Le polymorphisme a montré dans cet exemple ses extraordinaires facultés d'adaptation et de réutilisation. Nous avons constaté le gain en effort de développement obtenu par l'équipe qui a choisi de réfléchir à la construction d'une hiérarchie fondée sur une classe abstraite. Nous conseillons donc de penser à la notion de **classe abstraite** et aux **méthodes virtuelles** lors de la programmation d'un problème avec des classes.

5.4 Programmation événementielle et visuelle

Plan du chapitre:

Introduction

Programmation visuelle basée sur les pictogrammes
Programmation orientée événements
Normalisation du graphe événementiel

le graphe événementiel *arcs et sommets*
les diagrammes d'états UML *réduits*

Tableau des actions événementielles
Interfaces liées à un graphe événementiel
Avantages et modèle de développement RAD visuel
le modèle de la spirale *(B.Boehm)*
le modèle incrémental

1. Programmation visuelle basée sur les pictogrammes

Le développement visuel rapide d'application est fondé sur le concept de programmation visuelle associée à la montée en puissance de l'utilisation des Interactions Homme-Machine (IHM) dont le dynamisme récent ne peut pas être méconnu surtout par le débutant. En informatique les systèmes MacOs, Windows, les navigateurs Web, sont les principaux acteurs de l'ingénierie de l'IHM. Actuellement dans le développement d'un logiciel, un temps très important est consacré à l'ergonomie et la communication, cette part ne pourra que grandir dans un avenir proche; car les utilisateurs veulent s'adresser à des logiciels efficaces (ce qui va de soi) mais aussi conviviaux et faciles d'accès.

Les développeurs ont donc besoin d'avoir à leur disposition des produits de développement adaptés aux nécessités du moment. A ce jour la programmation visuelle est une des réponses à cette attente des développeurs.

La programmation visuelle au tout début a été conçue pour des personnes n'étant pas des programmeurs en basant ses outils sur des manipulations de pictogrammes.
Le raisonnement communément admis est qu'un dessin associé à une action élémentaire est plus porteur de sens qu'une phrase de texte.

A titre d'exemple ci-dessous l'on enlève le "fichier.bmp" afin de l'effacer selon deux modes de communication avec la machine: utilisation d'icônes ou entrée d'une commande textuelle.

Effacement avec un langage d'action visuelle (souris)

Action :	**Réponse :**

Effacement avec un langage textuel (clavier)

Action :	**Réponse :**
del c:\Exemple\Fichier.bmp	| ?

Nous remarquons donc déjà que l'interface de communication MacOs, Windows dénommée "bureau électronique" est en fait un outil de programmation de commandes systèmes.

Un langage de programmation visuelle permet "d'écrire" la partie communication d'un programme uniquement avec des dessins, diagrammes, icônes etc... Nous nous intéressons aux systèmes RAD (Rapid Application Development) visuels, qui sont fondés sur des langages objets à bases d'icônes ou pictogrammes. Visual Basic de MicroSoft est le premier RAD visuel a avoir été commercialisé dès 1991, il est fondé sur un langage Basic étendu incluant des objets étendus en VB.Net depuis 2001, puis dès 1995 Delphi le premier RAD visuel de Borland fondé sur Pascal objet, puis actuellement toujours de Borland : C++Builder RAD visuel fondé sur le langage C++ et Jbuilder, NetBeans RAD visuel de Sun fondés sur le langage Java, Visual C++, Visual J++ de Microsoft, Visual C# etc...

Le développeur trouve actuellement, une offre importante en outil de développement de RAD visuel y compris en open source. Nous proposons de définir un langage de RAD visuel ainsi :

Un **langage visuel** dans un RAD visuel est un **générateur de code source** du langage de base qui, derrière chaque action visuelle (dépôt de contrôle, click de souris, modifications des propriétés, etc...) engendre des lignes de code automatiquement et d'un manière transparente au développeur.

Des outils de développement tels que Visual Basic ou Delphi sont adaptés à la programmation visuelle pour débutant. Toutefois l'efficacité des dernières versions depuis 3 ans a étendu leur champ au développement en général et dans l'activité industrielle et commerciale avec des versions "entreprise" pour VB et "Architect "pour Delphi.

En outre, le système windows est le plus largement répandu sur les machines grand public (90% des PC vendus en sont équipés), il est donc très utile que le débutant en programmation sache utiliser un produit de développement (rapide si possible) sur ce système.

Proposition :

Nous considérons dans cet ouvrage, la programmation visuelle à la fois comme une **fin** et comme un **moyen**.

La programmation visuelle est sous-tendue par la réactivité des programmes en réponse aux actions de l'utilisateur. Il est donc nécessaire de construire des programmes qui répondent à des sollicitations externes ou internes et non plus de programmer séquentiellement (ceci est essentiellement dû aux architectures de Von Neumann des machines) : ces sollicitations sont appelées des événements.

Le concept de **programmation dirigée ou orientée par les événements** est donc la composante essentielle de la programmation visuelle.

Terminons cette présentation par 5 remarques sur le concept de RAD :

Utiliser un RAD simple mais puissant

Nous ne considérerons pas comme utile pour des débutants de démarrer la programmtion visuelle avec des RAD basés sur le **langage C++**. Du fait de sa large permissivité ce langage permet au programmeur d'adopter certaines **attitudes dangereuses** sans contrôle possible. Seul le programmeur confirmé au courant des pièges et des subtilités de la programmation et du langage, pourra exploiter sans risque la richesse de ce type de RAD.

Avoir de bonnes méthodes dès le début

Le RAD Delphi de Borland conçu en 1995 est une extension du langage Object Pascal, qui a des **caractéristiques très proches de celles de C++** sans en avoir les **inconvénients**. L'aspect fortement typé du langage pascal autorise la prise en compte par le développeur débutant de bonnes attitudes de programmation.

C'est Apple qui en a été le promoteur

Le premier environnement de développement visuel professionnel basé sur **Object Pascal** a été conçu par Apple pour le système d'exploitation **MacOs**, sous la dénomination de **MacApp** en 1986. Cet environnement objet visuel permettait de développer des applications MacIntosh avec souris, fenêtre, menus déroulants etc...

Microsoft l'a popularisé

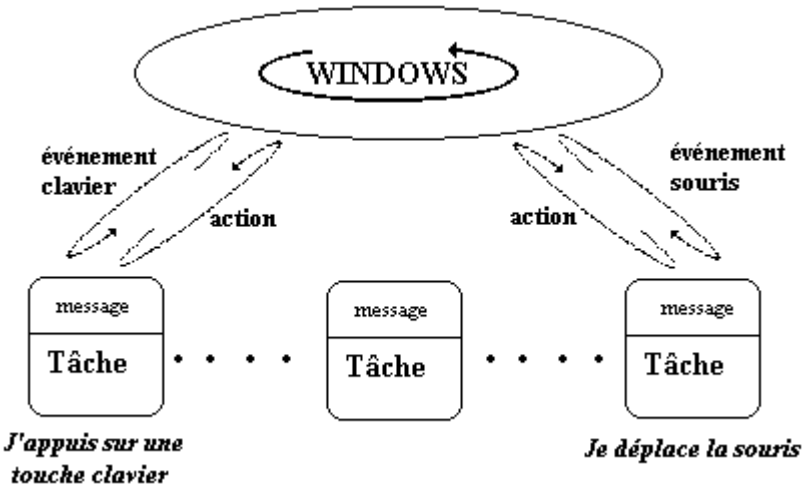
Le RAD Visual Basic de MicroSof conçu à partir de 1992, basé sur le langage Basic avait pour objectif le développement de petits logiciels sous Windows par des **programmeurs non expérimentés et occasionnels**. Actuellement il se décline en VB.Net un langage totalement orienté objet faisant partie intégrante de la plate-forme .Net Framework de Microsoft.

Le concept de RAD a de beaux jours devant lui

Le métier de développeur devrait à terme, consister grâce à des outils tels que les RAD visuels, à prendre un "caddie" et à aller dans un supermarché de composants logiciels génériques adaptés à son problème. Il ne lui resterait plus qu'à assembler le flot des événements reliant entre eux ces logiciels en kit.

2. Programmation orientée événements

Sous les versions actuelles de Windows, système multi-tâches préemptif sur micro-ordinateur, les concepts quant à la programmation par événement restent sensiblement les mêmes que sous les anciennes versions.



Nous dirons que le système d'exploitation passe l'essentiel de son " temps " à attendre une action de l'utilisateur (événement). Cette action déclenche un message que le système traite et envoie éventuellement à une application donnée.

Une définition de la programmation orientée événements

Logique de conception selon laquelle un programme est construit avec des objets et leurs propriétés et d'après laquelle les interventions de l'utilisateur sur les objets du programme déclenchent l'exécution des routines associées.

Par la suite, nous allons voir dans ce chapitre que la programmation d'une application " windows-like " est essentiellement une programmation par événements associée à une programmation classique.

Nous pourrons construire un logiciel qui réagira sur les interventions de l'utilisateur si nous arrivons à intercepter dans notre application les messages que le système envoie. Or l'environnement RAD (Delphi , comme d'ailleurs avant lui Visual Basic de Microsoft), autorise la consultation de tels messages d'un façon simple et souple.

Deux approches pour construire un programme

- *L'**approche événementielle*** intervient principalement dans l'interface entre le logiciel et l'utilisateur, mais aussi dans la liaison dynamique du logiciel avec le système, et enfin dans la sécurité.
- *L'**approche visuelle*** nous aide et simplifie notre tâche dans la construction du dialogue homme-machine.
- La combinaison de ces deux approches produit un logiciel habillé et adapté au système d'exploitation.

Il est possible de relier certains objets entre eux par des relations événementielles. Nous les représenterons par un graphe (structure classique utilisée pour représenter des relations). Lorsque l'on utilise un système multi-fenêtré du genre windows, l'on dispose du clavier et de la souris pour agir sur le système. En utilisant un RAD visuel, il est possible de **construire un logiciel qui se comporte comme le système sur lequel il s'exécute**. L'intérêt est que l'utilisateur aura moins d'efforts à accomplir pour se servir du programme puisqu'il aura des fonctionnalités semblables au système. Le fait que l'utilisateur reste dans un **environnement familier** au niveau de la manipulation et du confort du dialogue, assure le logiciel d'un capital confiance de départ non négligeable.

3. Normalisation du graphe événementiel

Il n'existe que peu d'éléments accessibles aux débutants sur la programmation orientée objet par événements. Nous construisons une démarche méthodique pour le débutant, en partant de remarques simples que nous décrivons sous forme de schémas dérivés des diagrammes d'états d'UML. Ces schémas seront utiles pour nous aider à décrire et à implanter des relations événementielles en Delphi ou dans un autre RAD événementiel.

Voici deux principes qui pour l'instant seront suffisants à nos activités de programmation.

Dans une interface windows-like nous savons que:

- Certains événements déclenchent immédiatement des actions comme par exemple des appels de routines système.
- D'autres événements ne déclenchent pas d'actions apparentes mais activent ou désactivent certains autres événements système.

Nous allons utiliser ces deux principes pour conduire notre programmation par événements.

Nous commencerons par **le concept d'activation et de désactivation**, les autres événements fonctionneront selon les mêmes bases. Dans un enseignement sur la programmation événementielle, nous avons constaté que ce **concept était suffisant** pour que les étudiants comprennent les fondamentaux de l'approche événementielle.

Remarque :

Attention! Il ne s'agit que d'une manière particulière de conduire notre programmation, ce ne peut donc être ni la seule, ni la meilleure (le sens accordé au mot meilleur est relatif au domaine pédagogique). Cette démarche s'est révélée être fructueuse lors d'enseignements d'initiation à ce genre de programmation.

Hypothèses de construction

Nous supposons donc que lorsque l'utilisateur intervient sur le programme en cours d'exécution, ce dernier réagira en première analyse de deux manières possibles :

- soit il lancera l'appel d'un routine (exécution d'une action, calcul, lecture de fichier, message à un autre objet comme ouverture d'une fiche etc...),
- soit il modifiera l'état d'activation d'autres objets du programme et/ou de lui-même, soit il ne se passera rien, nous dirons alors qu'il s'agit d'une modification nulle.

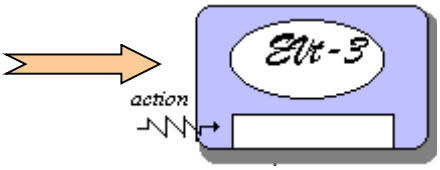
Ces hypothèses sont largement suffisantes pour la plupart des logiciels que nous pouvons raisonnablement espérer construire en initiation. Les concepts plus techniques de messages dépassent assez vite l'étudiant qui risque de replonger dans de " la grande bidouille ".

3.1 le graphe événementiel arcs et sommets

Nous proposons de construire un graphe dans lequel :

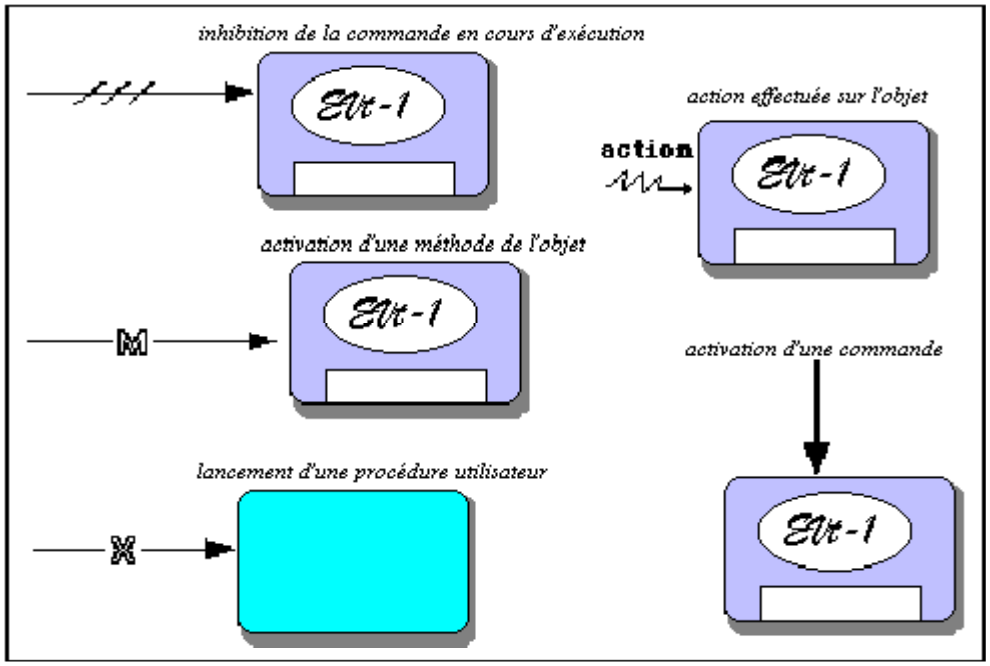
chaque **sommet** est un objet sensible à un événement donné.

L'événement donné est déclenché par une action extérieure à l'objet.

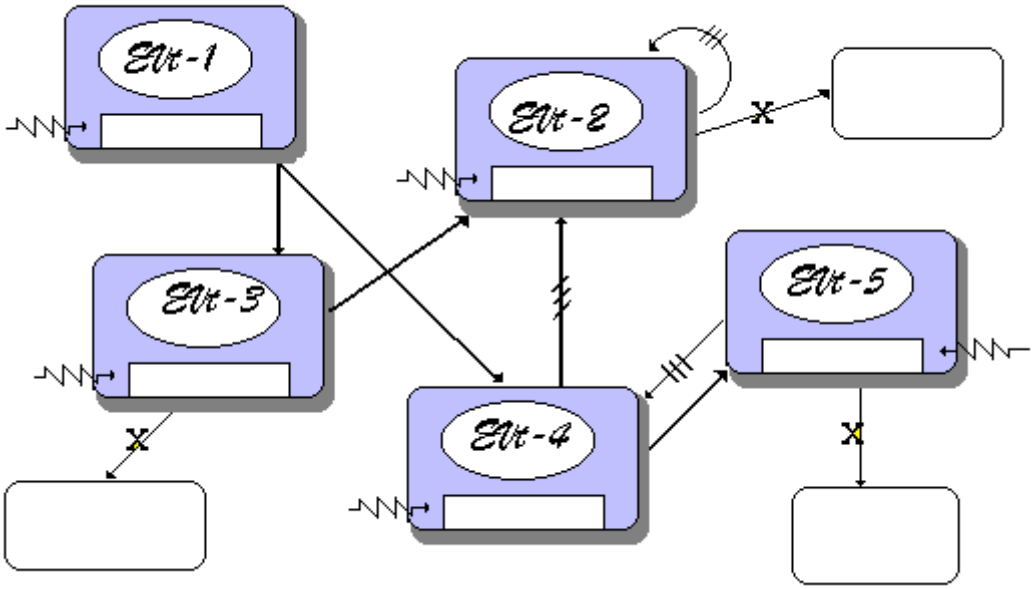


Les arcs du graphe représentent des actions lancées par un sommet.

Les actions sont de 4 types

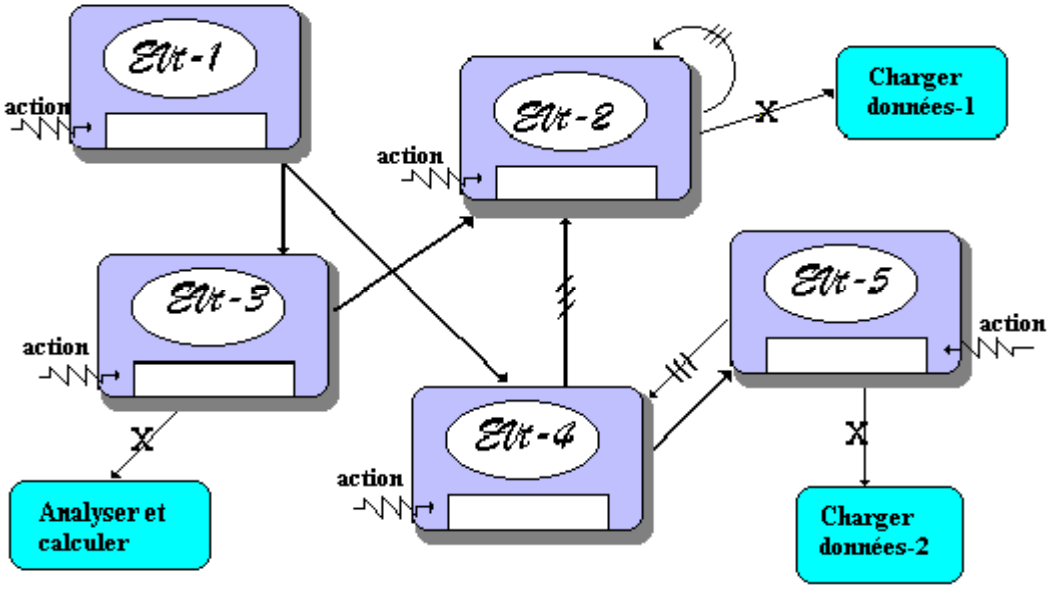


Soit le graphe événementiel suivant composé de 5 objets sensibles chacun à un événement particulier dénoté Evt-1,..., Evt-5; ce graphe comporte des réactions de chaque objet à l'événement auquel il est sensible :



Imaginons que ce graphe corresponde à une analyse de chargements de deux types différents de données à des fins de calcul sur leurs valeurs.

La figure suivante propose un tel graphe événementiel à partir du graphe vide précédent.



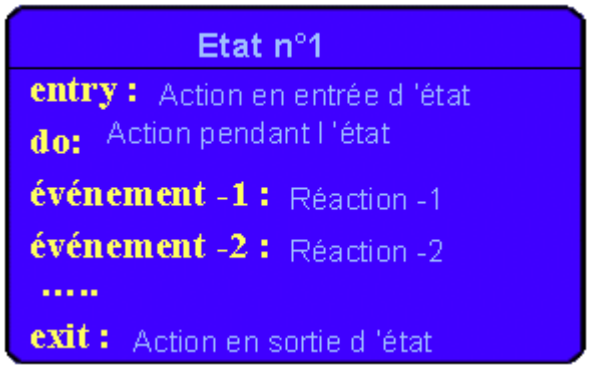
Cette notation de graphe événementiel est destinée à s'initier à la pratique de la description d'au maximum 4 types de réactions d'un objet sur la sollicitation d'un seul événement.

Remarques :

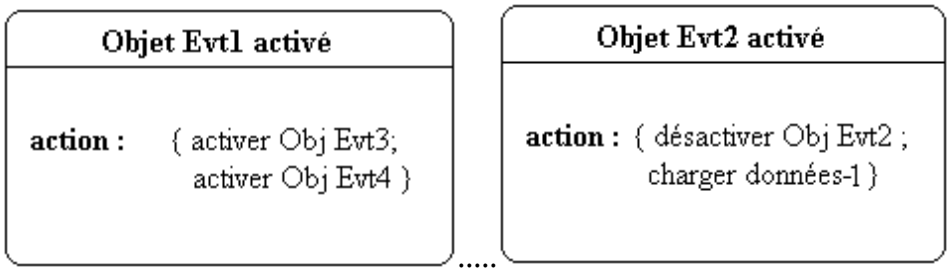
- L'arc nommé, représentant l'activation d'une méthode correspond très exactement à la notation UML de l'envoi d'un message.
- Lorsque nous voudrons représenter d'une manière plus complète d'autres réactions d'**un seul** objet **à plusieurs événements différents**, nous pourrons utiliser la notation UML réduite de diagramme d'état pour un objet (réduite parce qu'un objet **visuel** ne prendra pour nous, que 2 états: activé ou désactivé).

3.2 les diagrammes d'états UML réduits

Nous livrons ci-dessous la notation générale de diagramme d'état en UML, les cas particuliers et les détails complets sont décrits dans le document de spécification d'UML.



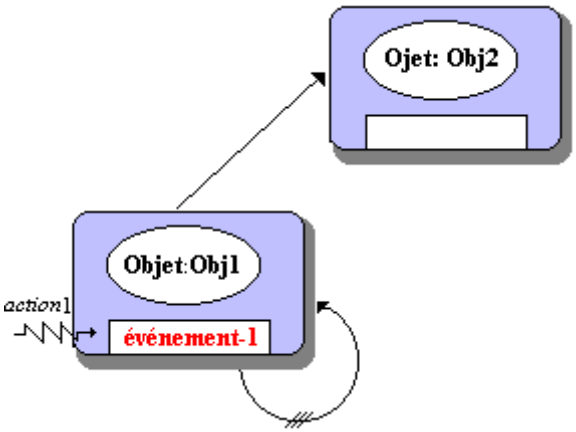
Voici les diagrammes d'états réduits extraits du graphe événementiel précédent, pour les objets **Evt-1** et **Evt-2** :



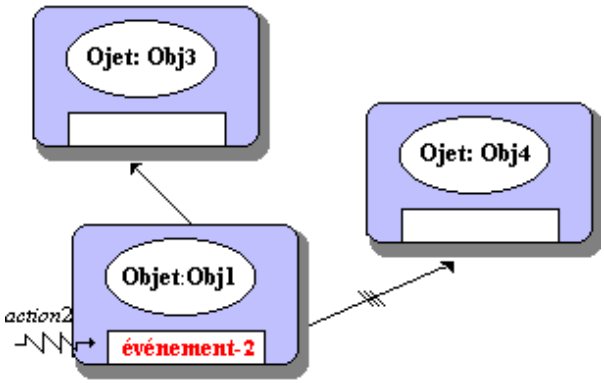
etc...

Nous remarquerons que cette écriture, pour l'instant, ne produit pas plus de sens que le graphe précédent qui comporte en sus la vision globale des interrelations entre les objets.
Ces diagrammes d'états réduits deviennent plus intéressants lorsque nous voulons exprimer le fait par exemple, qu'un seul objet **Obj1** réagit à 3 événements (événement-1, événement-2, événement-3). Dans ce cas décrivons les portions de graphe événementiel associés à chacun des événements :

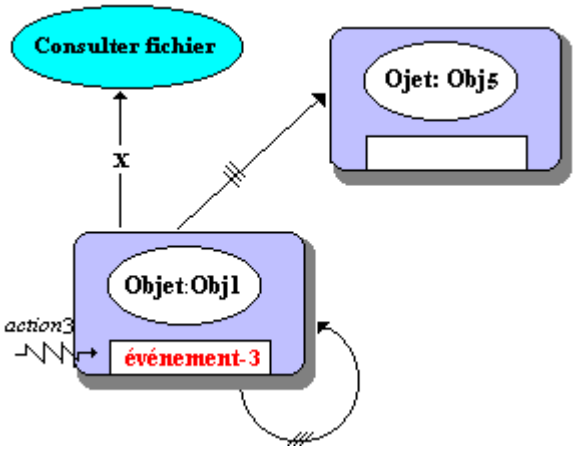
Réaction de obj1 à l'événement-1 :



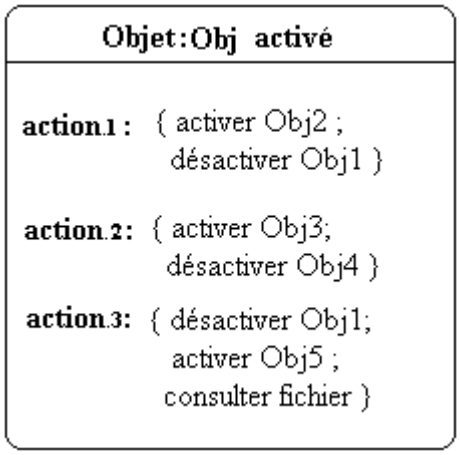
Réaction de obj1 à l'événement-2 :



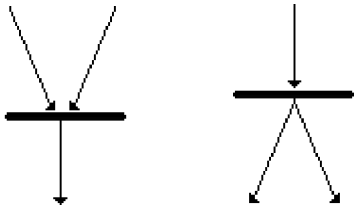
Réaction de obj1 à l'événement-3 :



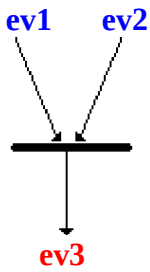
Synthétisons dans un diagramme d'état réduit les réactions à ces 3 événements :



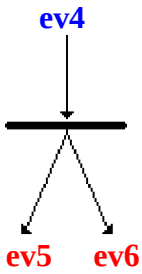
Lorsque nous jugerons nécessaire à la compréhension de relations événementielles dans un logiciel visuel, nous pourrons donc utiliser ce genre de diagramme pour renforcer la sémantique de conception des objets visuels. La notation UML sur les diagrammes d'états comprend les notions d'état de départ, de sortie, imbriqué, historisé, concurrents...
Lorsque cela sera nécessaire nous utiliserons la notation UML de synchronisation d'événements :



Dans le premier cas la notation représente la conjonction des deux événements **ev1** et **ev2** qui déclenche l'événement **ev3**.



Dans le second cas la notation représente l'événement **ev4** déclenchant conjointement les deux événements **ev5** et **ev6**.



4. Tableau des actions événementielles

L'exemple de graphe événementiel précédent correspond à une application qui serait sensible à 5 événements notés EVT-1 à EVT-5, et qui exécuterait 3 procédures utilisateur. Nous

notons dans un tableau (nommé " ***tableau des actions événementielles*** ")les résultats obtenus par analyse du graphe précédent, événement par événement..

EVT-1	EVT-3 activable EVT-4 activable
EVT-2	Appel de procédure utilisateur "chargement-1" désactivation de l' événement EVT-2
EVT-3	Appel de procédure utilisateur "Analyser" EVT-2 activable
EVT-4	EVT-2 désactivé EVT-5 activable
EVT-5	EVT-4 désactivé immédiatement Appel de procédure utilisateur "chargement-2"

Nous adjoignons à ce tableau une table des états des événements dès le lancement du logiciel (elle correspond à l'état initial des objets). Par exemple ici :

Evt1	activé
Evt2	désactivé
Evt3	activé
Evt4	activé
Evt5	désactivé

etc...

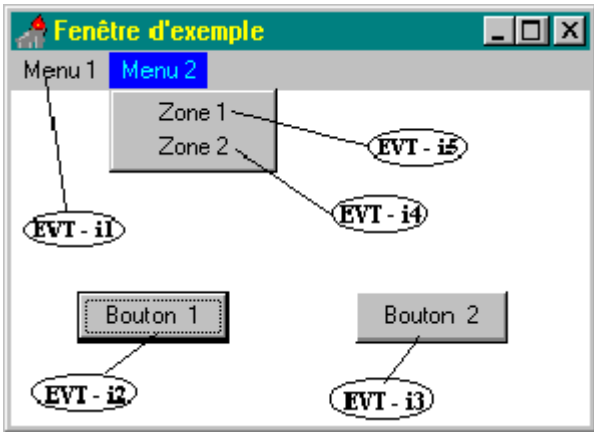
5. Interfaces liées à un graphe événementiel

construction d'interfaces liées au graphe précédent

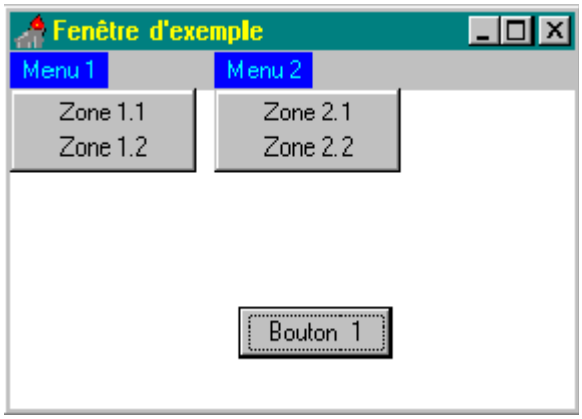
Dans l'exemple de droite, (i1,i2,i3,i4,i5)est une permutation sur (1,2,3,4,5) .

Ce qui nous donne déjà 5!=120 interfaces possibles avec ces objets et uniquement avec cette topologie.

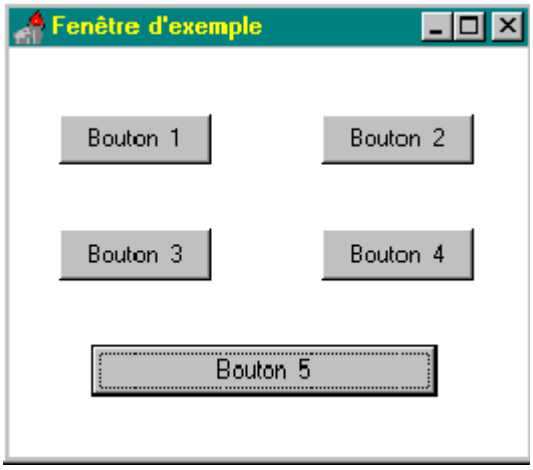
Interface n°1



La figure précédente montre une IHM (construite avec des objets Delphi) à partir du graphe événementiel étudié plus haut. Ci-dessous deux autres structures d'interfaces possibles avec les mêmes objets combinés différemment et associés au même graphe événementiel :



Interface n°2



Interface n°3

Pour les choix n°2 et n°3, il y a aussi 120 interfaces possibles…

6. Avantages du modèle de développement RAD visuel

L'approche uniquement structurée (privilégiant les fonctions du logiciel) impose d'écrire du code long et compliqué en risquant de ne pas aboutir, car il faut tout tester afin d'assurer un bon fonctionnement de tous les éléments du logiciel.

L'approche événementielle préfère bâtir un logiciel fondé sur une construction graduelle en fonction des besoins de communication entre l'humain et la machine. Dans cette optique, le programmeur élabore les fonctions associées à une action de communication en privilégiant le dialogue. Ainsi les actions internes du logiciel sont subordonnées au flux du dialogue.

Avantages liés à la programmation par RAD visuel

- Il est possible de construire très rapidement un prototype.
- Les fonctionnalités de communication sont les guides principaux du développement (approche plus vivante et attrayante).
- L'étudiant est impliqué immédiatement dans le processus de conception - construction.

L'étudiant acquiert très vite comme naturelle l'attitude de réutilisation en se servant de " logiciels en kit " (soit au début des composants visuels ou non, puis par la suite ses propres composants).

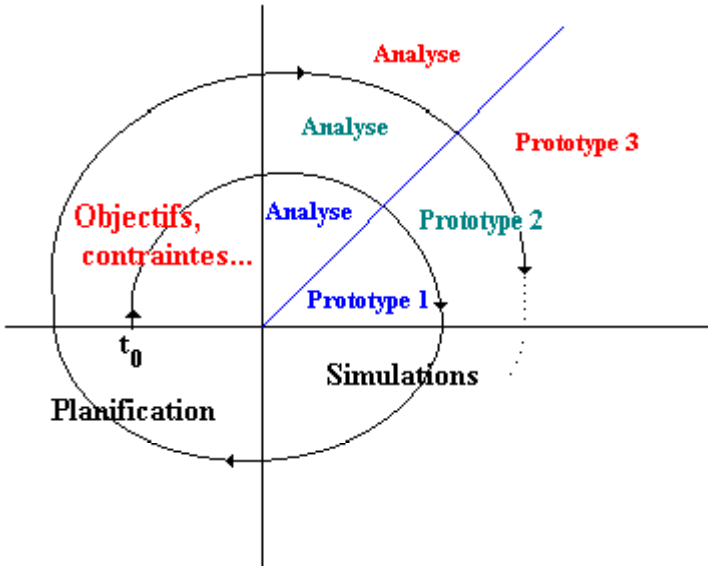
Il n'y a pas de conflit ni d'incohérence avec la démarche structurée : les algorithmes étant conçus comme des boîtes noires permettant d'implanter certaines actions, seront réutilisés immédiatement.
La méthodologie objet de COO et de POO reste un facteur d'intégration général des activités du programmeur.

Les actions de communications classiques sont assurées immédiatement par des objets standards (composants visuels ou non) réagissant à des événements extérieurs.
Le RAD fournira des classes d'objets standards non visuels (extensibles si l'étudiant augmente sa compétence) gérant les structures de données classiques (Liste, arbre, etc..).
L'extensibilité permet à l'enseignant de rajouter ses kits personnels d'objets et de les mettre à la disposition des étudiants comme des outils standards.

Modèles de développement avec un RAD visuel objet

Le développement avec ce genre de produit autorise une logique générale articulée sur la combinaison de deux modèles de développement :

le modèle de la spirale (B.Boehm) en version simplifiée

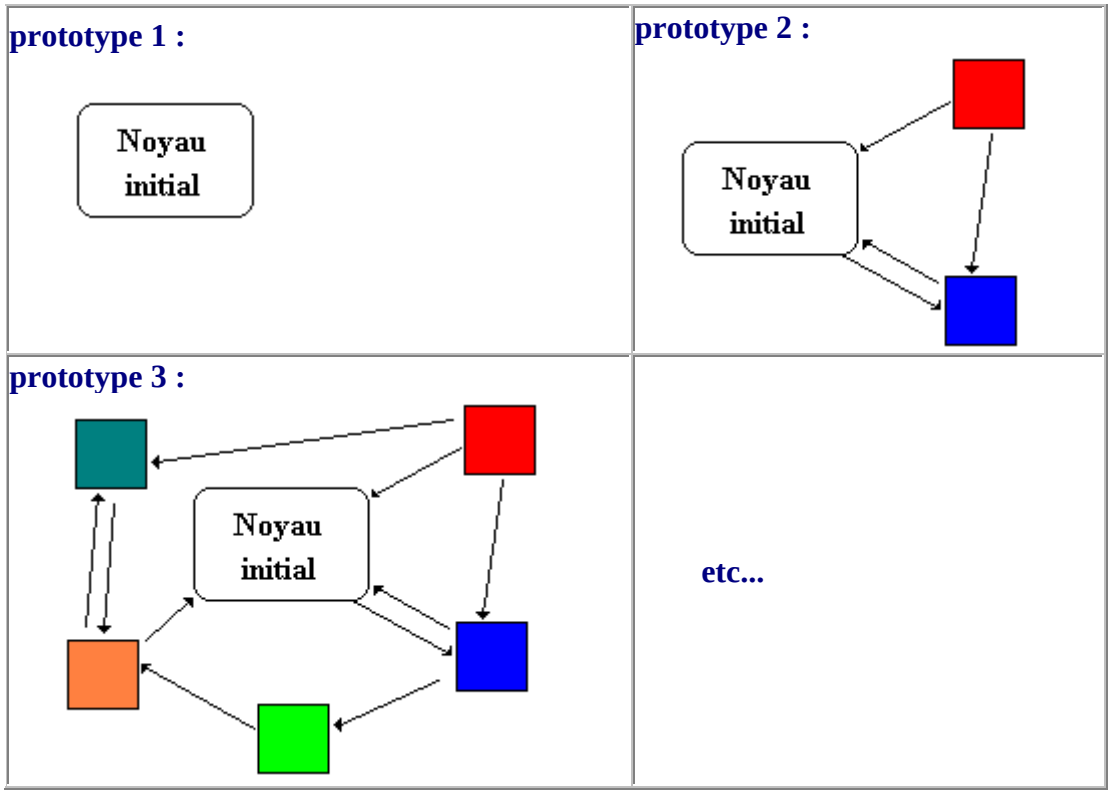


Dans le modèle de la spirale la programmation exploratoire est utilisée sous forme de prototypes simplifiés cycle après cycle. L'analyse s'améliore au cours de chaque cycle et fixe le type de développement pour ce tour de spirale.

le modèle incrémental

Il permet de réaliser chaque prototype avec un bloc central au départ, s'enrichissant à chaque phase de nouveaux composants et ainsi que de leurs interactions.

Associé à un cycle de prototypage dans la spirale, une seule famille de composants est développée pour un cycle fixé.



Ce modèle de développement à l'aide d'objets visuels ou non, fournira en fin de parcours un prototype opérationnel qui pourra s'intégrer dans un projet plus général.

Nous verrons sur des exemples comment ce type d'outil peut procurer aussi des avantages au niveau de la programmation défensive.

5.5 Les événements avec Delphi

Plan du chapitre:

Programmes événementiels

Pointeur de méthode
Affecter un pointeur de méthode
Un événement est un pointeur de méthode
Quel est le code engendré
Exercice-récapitulatif
Notice méthodologique pour créer un nouvel événement

1. Programmes événementiels avec Delphi

Delphi comme les autres RAD événementiels permet de construire du code qui est exécuté en réponse à des événements. Un événement Delphi est une propriété d'un type spécial que nous allons examiner plus loin.

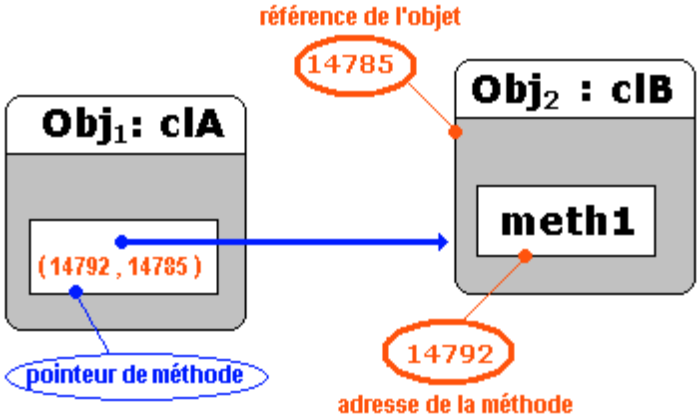
Le code de réaction à un événement particulier est une méthode qui s'appelle un *gestionnaire de cet événement*.

Un événement est en fait un pointeur vers la méthode qui est chargée de le gérer. Définissons la notion de pointeur de méthode qui est utilisée ici, notion largement utilisée en général dans les langages objets.

Pointeur de méthode

Un **pointeur de méthode** est **une paire de pointeurs** (adresses mémoire), le premier contient l'**adresse d'une méthode** et le **second une référence à l'objet** auquel appartient la méthode.

Schéma ci-après d'un objet **Obj1** de classe **clA** contenant un champ du type pointeur vers la méthode **meth1** de l'objet **Obj2** de classe **clB** :



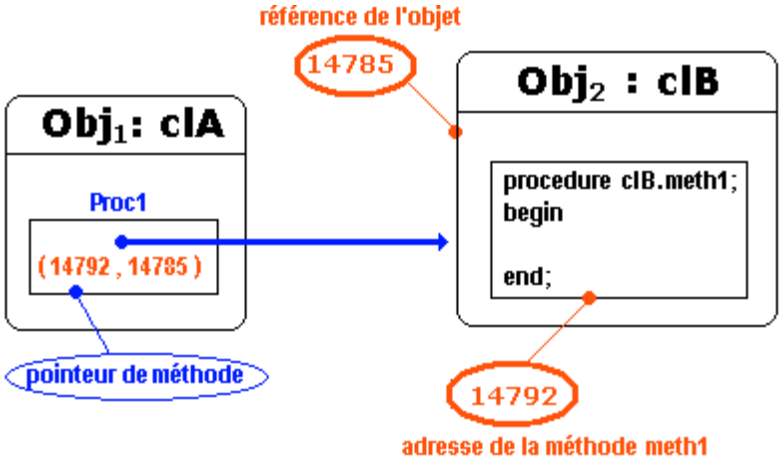
Pointeur de méthode en Delphi

Pour pointer la méthode d'une instance d'objet en Delphi, nous devons déclarer un nouveau type auquel nous ajoutons à la fin le qualificateur **of object** .

Exemples de types pointeurs de méthode et de variable de type pointeur de méthode :

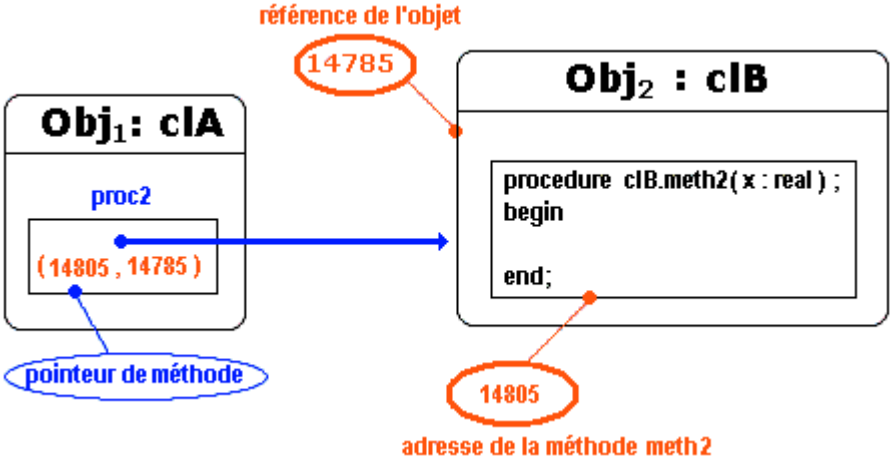
```
type

  ptrMethode1 = procedure of object;
  ptrMethode2 = procedure (x : real) of object;
  ptrMethode3 = procedure (x,y: integer; var z:char) of object;

var
    Proc1 : ptrMethode1;  // pointeur vers une méthode sans paramètre
    Proc2 : ptrMethode2; // pointeur vers une méthode à un  paramètre real
    Proc3 : ptrMethode3; // pointeur vers une méthode à trois  paramètres 2 entiers, 1 char
```

Schéma ci-après d'un objet **Obj1** de classe **clA** contenant un champ proc1 du type ptrMethode1 qui pointe vers la **meth1** de l'objet **Obj2** de classe **clB.** On suppose que l'adresse mémoire de l'objet est 14785 et l'adresse de la méthode **meth1** de l'objet est 14792 :



Il est impératif que la méthode vers laquelle pointe la variable de pointeur de méthode soit du type prévu par le type pointeur de méthode, ici la méthode meth1 doit obligatoirement être une procédure sans paramètre (compatibilité d'en-tête).

Schéma ci-après d'un objet **Obj1** de classe **clA** contenant un champ proc2 du type ptrMethode2 qui pointe vers la **meth2** de l'objet **Obj2** de classe **clB.** On suppose que l'adresse mémoire de l'objet est 14785 et l'adresse de la méthode **meth2** de l'objet est 14805 :



La remarque précédente sur l'obligation de compatibilité de l'en-tête de la méthode et le type pointeur de méthode implique que la méthode **meth2** doit nécessairement être une procédure à un seul paramètre real.

Affecter un pointeur de méthode

Nous venons de voir comment déclarer un type pointeur de méthode et un champ de ce même type, nous avons signalé que ce champ doit pointer vers une méthode ayant une en-tête compatible avec le type pointeur de méthode, il nous reste à connaître le mécanisme qu'utilise Delphi pour **lier un champ pointeur et une méthode à pointer**.

Types pointeurs de méthodes

ptrMethode1 = **procedure of object**;
ptrMethode2 = **procedure** (x : real) **of object**;

champs de pointeurs de méthodes
Proc1 : ptrMethode1; *// pointeur vers une méthode sans paramètre*
Proc2 : ptrMethode2; *// pointeur vers une méthode à un paramètre real*

Diverses méthodes :

procedure P1;
begin
 	
end;
	procedure P2 (x:real);
	begin
	
	end;
		procedure P3 (x:char);
		begin
		
		end;
			procedure P4;
			begin
			
			end;

Recensons d'abord les compatibilités d'en-tête qui autoriseront le pointage de la méthode :
Proc1 peut pointer vers P1 , P4 qui sont les deux seules méthodes compatibles.
Proc2 ne peut pointer que vers P2 qui est la seule méthode à un paramètre de type real.

La liaison (le pointage) s'effectue tout naturellement à travers une affectation :
L'affectation Proc1 := P1; indique que Proc1 pointe maintenant vers la méthode P1 et peut être utilisé comme un identificateur de procédure ayant la même signature que P1.

Exemple d'utilisation :

Proc2 := P2; *// liaison du pointeur et de la procédure P2*

Proc2(45.8); *// appel de la procédure vers laquelle Proc2 pointe avec passage du paramètre 45.8*

Un événement est un pointeur de méthode

Nous avons indiqué que les gestionnaires d'événements sont des méthodes, les champs du genre événements présents dans les classes Delphi sont en fait des pointeurs de méthode, qui peuvent pointer vers des gestionnaires d'événements.

Un type d'événement est donc un type pointeur de méthode, Delphi possède plusieurs types d'événements par exemple :

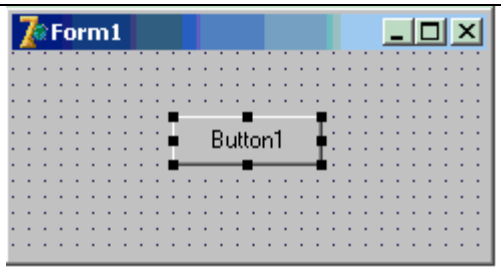
```
TNotifyEvent = procedure (Sender: TObject) of object;
TMouseMoveEvent = procedure (Sender: TObject; Shift: TShiftState; X, Y: Integer) of object;
TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
etc …
```

Un événement est donc une propriété de type pointeur de méthode (type événement) :

```
property OnClick: TNotifyEvent;          // événement click de souris
property OnMouseMove: TMouseMoveEvent; // événement passage de la souris
property OnKeyPress: TKeyPressEvent; // événement touche de clavier pressée
```

Quel est le code engendré pour gérer un événement ?

Intéressons nous maintenant au code engendré par un programme simple constitué d'une fiche Form1 de classe TForm1 avec un objet Button1 de classe Tbutton déposé sur la fiche :




```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms,   Dialogs, StdCtrls;

type
 TForm1 = class(TForm)
   Button1: TButton;
 private
   { Déclarations privées }
 public
   { Déclarations publiques }
 end;

var   Form1: TForm1;

implementation

{$R *.dfm}

end.
```

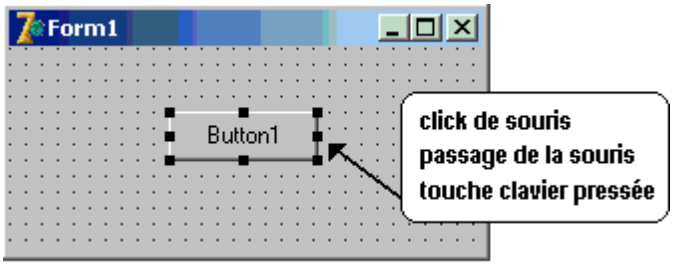
Le code source apparent fournit au programmeur. Il permet l'intervention du programmeur sur la fiche et sur ses composants.

```
object Form1: TForm1
  Left = 198
  Top = 109
  Width = 245
  Height = 130
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = [ ]
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13

  object Button1: TButton
    Left = 80
    Top = 32
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end

end
```

Le code intermédiaire caché au programmeur. Il permet l'initialisation automatique de la fiche et de ses composants.

Le code intermédiaire caché de l'objet **Button1** déposé sur la fiche Form1.

Demandons à Delphi de nous fournir (à partir de l'inspecteur d'objet) les gestionnaires des 3 événements OnClick, OnMouseMove et OnKeyPress de réaction de l'objet Button1 au click de souris, au passage de la souris et à l'appui sur une touche du clavier:




```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms,   Dialogs, StdCtrls;

type
 TForm1 = class(TForm)
  Button1: TButton;
   procedure Button1Click (Sender: TObject);
   procedure Button1MouseMove (Sender: TObject;  Shift:  TShiftState; X,Y: Integer);
   procedure Button1KeyPress (Sender: TObject;    var Key: Char);

  private
   { Déclarations privées }
  public
   { Déclarations publiques }
  end;

var   Form1: TForm1;

implementation

{$R *.dfm}
procedure TForm1.Button1Click (Sender: TObject);
begin
                    { Gestionnaire OnClick }
end;

procedure TForm1.Button1MouseMove (Sender: TObject;  Shift:  TShiftState; X,Y: Integer);
begin
                    { Gestionnaire OnMouseMove }
end;

procedure TForm1.Button1KeyPress (Sender: TObject;    var Key: Char);
begin
                    { Gestionnaire OnKeyPress }
end;

end.
```

Nouveau code intermédiaire caché de l'objet **Button1** contenant les affectations des événéments à leurs gestionnaires.

```
object Button1: TButton
   Left = 80
   Top = 32
   Width = 75
   Height = 25
   Caption = 'Button1'
   TabOrder = 0
   OnClick = Button1Click
   OnKeyPress = Button1KeyPress
   OnMouseMove = Button1MouseMove
end
```

Delphi engendre un code source visible en pascal objet modifiable et un code intermédiaire (que l'on peut voir et éventuellement modifier) :

- Le code source visible est celui qui nous sert à programmer nos algorithmes, nos classes, et les réactions aux événements.

- Le code intermédiaire est généré au fur et à mesure que nous intervenons visuellement sur l'interface et contient l'initialisation de l'interface (affectations de gestionnaires d'événements, couleurs des fonds, positions des composants, taille, polices de caractères, conteneurs et contenus etc…) il sert au compilateur.

Nous venons de voir que Delphi a généré en code intermédiaire pour nous, les affectations de chaque événement à un gestionnaire :

```
OnClick = Button1Click
OnKeyPress = Button1KeyPress
OnMouseMove = Button1MouseMove
```

Et il nous a fournit les squelettes vides de chacun des trois gestionnaires :

```
procedure TForm1.Button1Click (Sender: TObject); …

procedure TForm1.Button1MouseMove (Sender: TObject;  Shift:  TShiftState; X,Y: Integer); …

procedure TForm1.Button1KeyPress (Sender: TObject;    var Key: Char); …
```

La dernière étape du processus de programmation de la réaction du Button1 est de programmer du code à l'intérieur des squelettes des gestionnaires.

Lors de l'exécution si nous cliquons avec la souris sur le Button1, un mécanisme d'interception et de répartition figuré ci-dessous appelle le gestionnaire de l'événement OnClick dont le corps a été programmé :

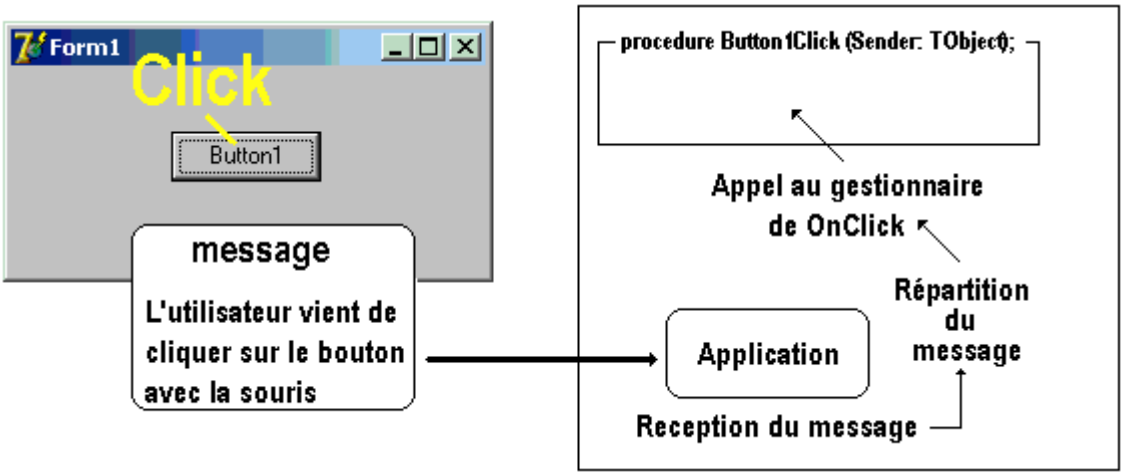


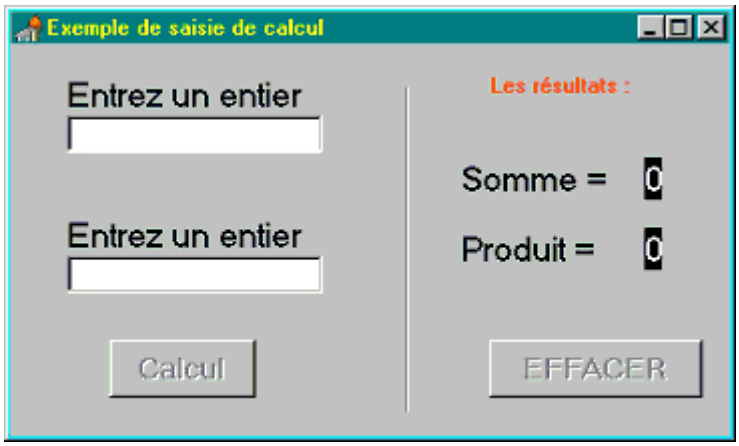
fig : *Appel du gestionnaire* ***procedure*** *TForm1.Button1Click (Sender: TObject) sur click de souris*

Exercice-récapitulatif

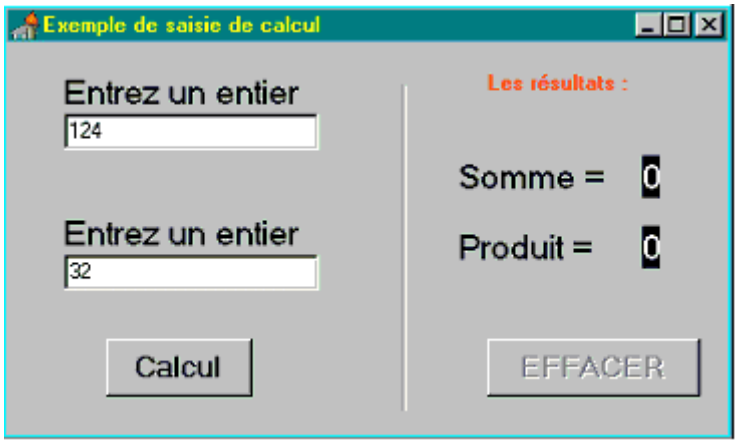
Nous voulons construire une interface permettant la saisie par l'utilisateur de deux entiers et l'autorisant à effectuer leur somme et leur produit uniquement lorsque les entiers sont entrés tous les deux. Aucune sécurité n'est apportée **pour l'instant** sur les données.

Voici l'état visuel de l'interface au lancement du programme :

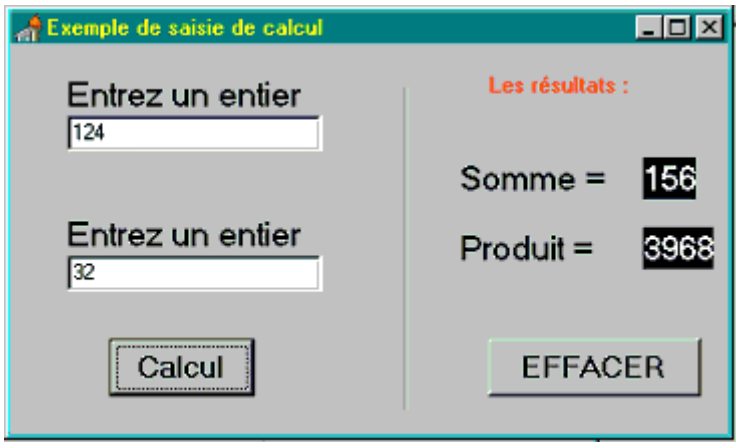
2 zones de saisie
2 boutons d'actions
2 zones d'affichages des résultats



Dès que les deux entiers sont entrés, le bouton **Calcul** est activé:



Lorsque l'on clique sur le bouton **Calcul**, le bouton **EFFACER** est activé et les résultats s'affichent dans leurs zones respectives :



Un clic sur le bouton **EFFACER** ramène l'interface à l'état initial.

Graphe événementiel complet de l'interface

Avec comme conventions sur les messages :

X_1 = exécuter les calculs sur les valeurs entrées.

X_2 = exécuter l'effacement des résultats.

M_1 = message à l'edit1 de tout effacer.

M_2 = message à l'edit2 de tout effacer.

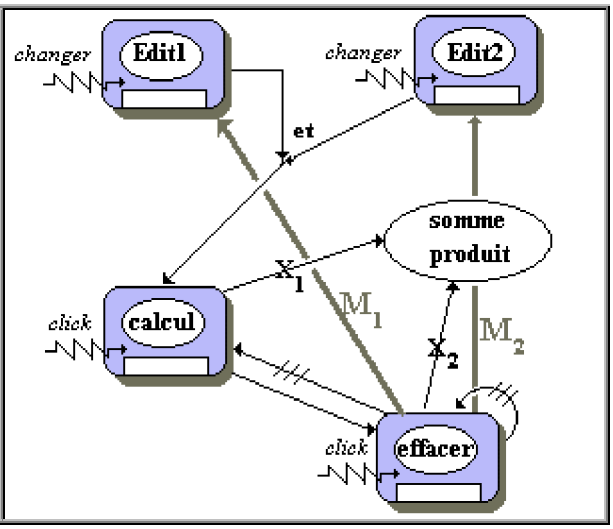


Table des actions événementielles associées au graphe

Changer Edit1	Calcul activable (si changer Edit2 a eu lieu)
Changer EDIT2	Calcul activable (si changer Edit1 a eu lieu)
Clic CALCUL	Exécuter X1 EFFACER activable Afficher les résultats
Clic EFFACER	EFFACER désactivé CALCUL désactivé Message M1 Message M2

Table des états initiaux des objets sensibles aux événements

Edit1	***activé***
Edit2	***activé***
Buttoncalcul	***désactivé***
Buttoneffacer	***désactivé***

- Nous voulons disposer d'un bouton permettant de lancer le calcul lorsque c'est possible et d'un bouton permettant de tout effacer. Les deux boutons seront désactivés au départ.
- Nous choisissons 6 objets visuels : deux TEdit, **Edit1** et **Edit2** pour la saisie ; deux Tbutton, **ButtonCalcul** et **Buttoneffacer** pour les changements de plans d'action ; deux Tlabel **LabelSomme** et **LabelProduit** pour les résultats

Les objets visuels Delphi choisis

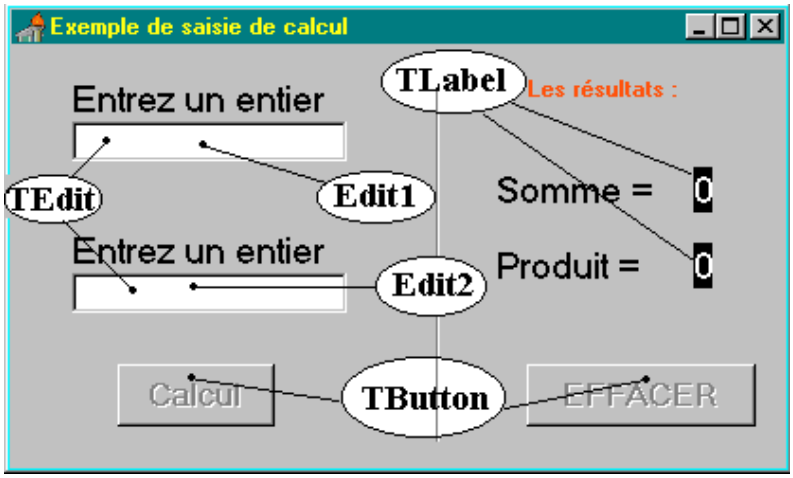
Edit1: TEdit;
Edit2: TEdit;
LabelSomme: TLabel;
LabelProduit: TLabel;

ButtonCalcul: TButton;

Calcul

Buttoneffacer: TButton;

EFFACER



Construction progressive du gestionnaire d'événement Onchange

Nous devons réaliser une synchronisation des entrées dans Edit1 et Edit2 : le déverrouillage (activation) du bouton " ButtonCalcul " ne doit avoir lieu que lorsque Edit1 et Edit2 contiennent des valeurs. Ces deux objets de classe TEdit, sont sensibles à **l'événement OnChange** qui indique que le contenu du champ **text** a été modifié, l'action est indiquée dans le graphe événementiel sous le vocable " *changer* ".

La synchronisation se fera à l'aide de deux drapeaux binaires (des booléens) qui seront levés chacun séparément par les TEdit lors du changement de leur contenu. Le drapeau Som_ok est levé par l'Edit1, le drapeau Prod_ok est levé par l'Edit2.

Implantation : deux champs privés booléens

```
Som_ok , Prod_ok : boolean;
```

Le drapeau Som_ok est levé par l'Edit1 lors du changement du contenu de son champ text, sur l'apparition de l'événement OnChange, il en est de même pour le drapeau Prod_ok et l'Edit2 :

Implantation n°1 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Som_ok:=true; // drapeau de Edit1 levé
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  Prod_ok:=true; // drapeau de Edit2 levé
end;
```

Une méthode privée de test vérifiera que les deux drapeaux ont été levés et lancera l'activation du **ButtonCalcul**.

```
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
 if  Prod_ok and Som_ok then
    ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;
```

Nous devons maintenant tester lorsque nous levons un drapeau si l'autre n'est pas déjà levé. Cette opération s'effectue dans les gestionnaires de l'événement OnChange de chacun des Edit1 et Edit2 :

Implantation n°2 du gestionnaire de OnChange :

procedure TForm1.Edit1Change(Sender: TObject); **begin** Som_ok:=true; *// drapeau de Edit1 levé* TestEntrees; **end**;	**procedure** TForm1.Edit2Change(Sender: TObject); **begin** Prod_ok:=true; *// drapeau de Edit2 levé* TestEntrees; **end**;

Construction des gestionnaires d'événement Onclick

Nous devons gérer l'événement clic sur le bouton calcul qui doit calculer la somme et le produit, placer les résultats dans les deux Tlabels et activer le Buttoneffacer.

Implantation du gestionnaire de OnClick du ButtonCalcul :

```
procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
    S:=strtoint(Edit1.text); // transtypage : string à integer
    P:=strtoint(Edit2.text); // transtypage : string à integer
    LabelSomme.caption:=inttostr(P+S);
    LabelProduit.caption:=inttostr(P*S);
    Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

Nous codons une méthode privée dont le rôle est de réinitialiser l'interface à son état de départ indiqué dans la table des états initiaux :

Edit1	***Activé***	**procedure** TForm1.RAZTout;
		begin
		Buttoneffacer.Enabled:=false; *//le bouton effacer se désactive*
		ButtonCalcul.Enabled:=false; *//le bouton calcul se désactive*
		LabelSomme.caption:='0'; *// RAZ valeur somme affichée*
		LabelProduit.caption:='0'; *// RAZ valeur produit affichée*
		Edit1.clear; *// message M1*
		Edit2.clear; *// message M2*
		Prod_ok:=false; *// RAZ drapeau Edit2*
		Som_ok:=false; *// RAZ drapeau Edit1*
		end;

Edit2	***Activé***	
Buttoncalcul	***Désactivé***	
Buttoneffacer	***désactivé***	

Nous devons gérer l'événement click sur le Buttoneffacer qui doit remettre l'interface à son état initial (par appel à la procédure RAZTout) :

Implantation du gestionnaire de OnClick du Buttoneffacer:

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;
```

Au final, lorsque l'application se lance et que l'interface est créée nous devons positionner tous les objets à l'état initial (nous choisissons de lancer cette initialisation sur l'événement **OnCreate** de création de la fiche):

Implantation du gestionnaire de OnCreate de la fiche Form1:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RAZTout;
end;
```

Si nous essayons notre interface nous constatons que nous avons un problème de sécurité à deux niveaux dans notre saisie :

- Le premier niveau est celui des corrections apportées par l'utilisateur (effacement de chiffres déjà entrés) et la synchronisation avec l'éventuelle vacuité d'au moins un des champs text après une telle modification : en effet l'utilisateur peut effacer tout le contenu d'un " Edit " et lancer alors le calcul, provoquant ainsi des erreurs.

- Le deuxième niveau classique est celui du transtypage incorrect, dans le cas où l'utilisateur commet des fautes de frappe et rentre des données autres que des chiffres (des lettres ou d'autres caractères du clavier).

Améliorations de sécurité du premier niveau par plan d'action

Nous protégeons les calculs dans le logiciel, par un test sur la vacuité du champ text de chaque objet de saisie TEdit. Dans le cas favorable où le champ n'est pas vide, on autorise la saisie ; dans l'autre cas on désactive systématiquement le ButtonCalcul et l'on abaisse le drapeau qui avait été levé lors de l'entrée du premier chiffre, ce qui empêchera toute erreur ultérieure. L'utilisateur comprendra de lui-même que tant qu'il n'y a pas de valeur dans les entrées, le logiciel ne fera rien et on ne passera donc pas au plan d'action suivant (calcul et affichage). Cette amélioration s'effectue dans les gestionnaires d'événement OnChange des deux TEdit.

Implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit1.text<' ' then // champ text non vide ok !
 begin
   Som_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; // bouton désactivé
   Som_ok:=false; // drapeau de Edit1 baissé
 end
end;
```

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit2.text<' ' then // champ text non vide ok !
 begin
   Prod_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
   Prod_ok:=false; // drapeau de Edit2 baissé
 end
end;
```

On remarque que les deux codes précédents sont très proches, ils diffèrent par leTEdit et le drapeau auxquels ils s'appliquent. Il est possible de réduire le code redondant en construisant par exemple une méthode privée avec deux paramètres.

Regroupement du code

Soit la méthode privée Autorise possédant deux paramètres, le premier est la référence d'un des deux TEdit, le second le drapeau booléen associé :

```
procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
 if Ed.text<' ' then // champ text non vide ok !
 begin
   flag :=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
    flag:=false; // drapeau de Ed baissé
 end
end;
```

Nouvelle implantation n°2 du gestionnaire de OnChange :

procedure TForm1.Edit1Change(Sender: TObject); **begin** Autorise (Edit1 , Som_ok); **end**;	**procedure** TForm1.Edit2Change(Sender: TObject); **begin** Autorise (Edit2 , Prod_ok); **end**;

Code final où tout est regroupé dans la classe TForm1

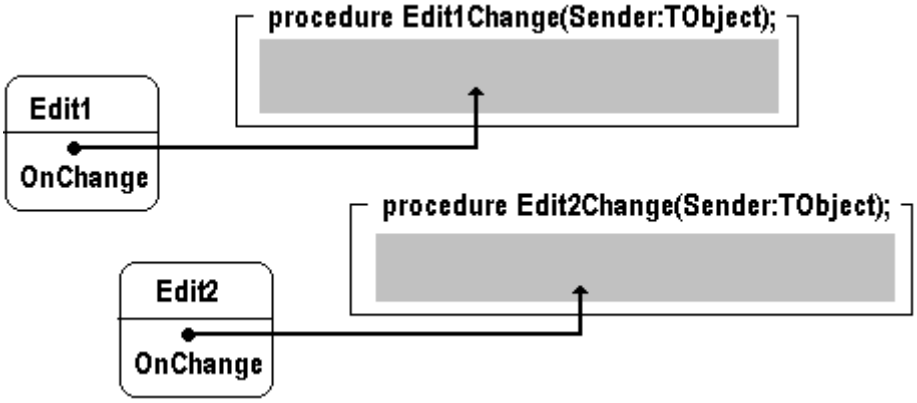
```
unit UFcalcul;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls, ExtCtrls;

type
 TForm1= class ( TForm )
    Edit1: TEdit;
    Edit2: TEdit;
    ButtonCalcul: TButton;
    LabelSomme: TLabel;
    LabelProduit: TLabel;
    Buttoneffacer: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure Edit2Change(Sender: TObject);
    procedure ButtonCalculClick(Sender: TObject);
    procedure ButtoneffacerClick(Sender: TObject);
  private   { Déclarations privées }
      Som_ok , Prod_ok :
      procedure TestEntrees;
      procedure RAZTout;
      procedure Autorise ( Ed : TEdit ; var flag : boolean );
  public   { Déclarations publiques }
  end;
```

```
implementation
{ --------------- Méthodes privées ---------------------}
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
  if  Prod_ok and Som_ok then
     ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;

procedure TForm1.RAZTout;
begin
    Buttoneffacer.Enabled:=false; //le bouton effacer se désactive
    ButtonCalcul.Enabled:=false; //le bouton calcul se désactive
    LabelSomme.caption:='0'; // RAZ valeur somme affichée
    LabelProduit.caption:='0'; // RAZ valeur produit affichée
    Edit1.clear; // message M1
    Edit2.clear; // message M2
    Prod_ok:=false; // RAZ drapeau Edit2
    Som_ok:=false; // RAZ drapeau Edit1
end;

procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
  if Ed.text<' ' then // champ text non vide ok !
  begin
    flag :=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; //bouton désactivé
     flag:=false; // drapeau de Ed baissé
  end
end;

{ ------------------- Gestionnaires d'événements -------------------}
procedure TForm1.FormCreate(Sender: TObject);
begin
   RAZTout;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
   Autorise ( Edit1 , Som_ok );
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
   Autorise ( Edit2 , Prod_ok );
end;

procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
     S:=strtoint(Edit1.text); // transtypage : string à integer
     P:=strtoint(Edit2.text); // transtypage : string à integer
     LabelSomme.caption:=inttostr(P+S);
     LabelProduit.caption:=inttostr(P*S);
     Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;

end.
```

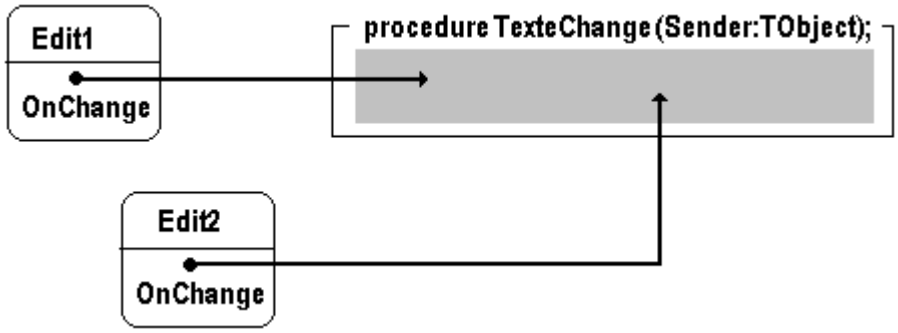
Un gestionnaire d'événement centralisé grâce au :
- **polymorphisme d'objet**
- **pointeur de méthode**

Chaque Edit possède son propre gestionnaire de l'événement OnChange :



procedure TForm1.Edit1Change(Sender: TObject); **begin** Autorise (Edit1 , Som_ok); **end**;	**procedure** TForm1.Edit2Change(Sender: TObject); **begin** Autorise (Edit2 , Prod_ok); **end**;

On peut avoir envie de mettre en place un gestionnaire unique de l'événement OnChange, puis de le lier à chaque zone de saisie Edit1 et Edit2 (souvenons-nous qu'un champ d'événement est un pointeur de méthode) :



Nous définissons d'abord une méthode public par exemple, qui jouera le rôle de gestionnaire centralisé d'événement, nous la nommons TexteChange.

Cette méthode pour être considérée comme un gestionnaire de l'événement OnChange, doit obligatoirement être compatible avec le type de l'événement Onchange. Une consultation de la documentation Delphi nous indique que :

property OnChange: TNotifyEvent;

L'événement OnChange est du même type que l'événement OnClick (TnotifyEvent = **procedure**(Sender:Tobject) **of object**), donc notre gestionnaire centralisé TexteChange doit avoir l'en-tête suivante :

procedure TexteChange (Sender : Tobject);

Le paramètre Sender est la référence de l'objet qui appelle la méthode qui est passé automatiquement lors de l'exécution, en l'occurrence ici lorsque Edit1 appellera ce gestionnaire c'est la référence de Edit1 qui sera passée comme paramètre, de même lorsqu'il s'agira d'un appel de Edit2.

Implantation du gestionnaire centralisé

```
procedure TForm1.TexteChange(Sender: TObject);
begin
  if  Sender is TEdit then
  begin
     if (Sender as TEdit )=Edit1 then
        Autorise ( (Sender as TEdit ) , Som_ok );
     else
        Autorise ( (Sender as TEdit ) , Prod_ok );
   end
end;
```

On teste si l'émetteur Sender est bien un TEdit,

polymorphisme d'objet :

Si l'émetteur est Edit1 on le **transtype** en TEdit pour pouvoir le passer en premier paramètre à la méthode Autorise **sinon** c'est Edit2 et l'on fait la même opération.

On lie maintenant ce gestionnaire à chacun des champs OnChange de chaque TEdit dès la création de la fiche :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RAZTout;
  Edit1.OnChange := TexteChange ;
  Edit2.OnChange := TexteChange ;
end;
```

pointeur de méthode

chaque champ Onchange pointe vers le même gestionnaire (méthode)

```
TForm1= class ( TForm )
   Edit1: TEdit;      …
   procedure FormCreate(Sender: TObject);
   procedure ButtonCalculClick(Sender: TObject);
   procedure ButtoneffacerClick(Sender: TObject);
  private    { Déclarations privées }
     Som_ok , Prod_ok :
     procedure TestEntrees;
     procedure RAZTout;
     procedure Autorise ( Ed : TEdit ; var flag : boolean );
  public    { Déclarations publiques }
    procedure TexteChange(Sender: TObject);
  end;
```

Le gestionnaire centralisé est déclaré ici, puis il est implémenté à la section **implementation** de la **unit**.

NOTICE METHODOLOGIQUE
construire un nouvel événement

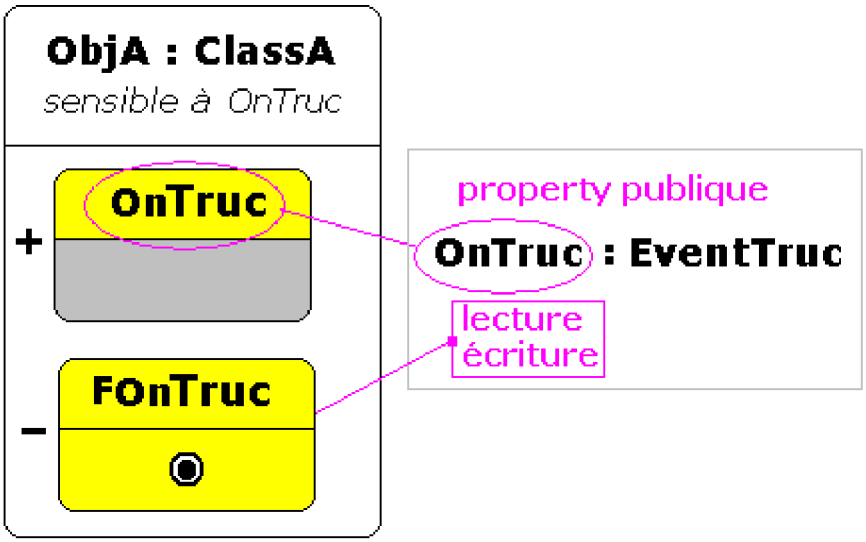
Objectif : Nous proposons ici en suivant pas à pas l'enrichissement du code de montrer comment implanter un nouvel événement nommé OnTruc dans une classe dénotée ClassA. Puis nous appliquerons cette démarche à une pile Lifo qui sera rendu sensible à l'empilement et au dépilement, par adjonction de deux événements à la classe.

Pour construire un nouvel événement dans ClassA :

- Il nous faut d'abord définir un type pour l'événement : EventTruc
- Il faut ensuite mettre dans ClassA une propriété d'événement : **property** OnTruc : EventTruc
- Il faut créer un champ privé nommé FOnTruc de type EventTruc en lecture et écriture qui servira de champ de stockage de la propriété OnTruc.

type de l'événement : Pointeur de méthode

EventTruc = procedure (...) of object

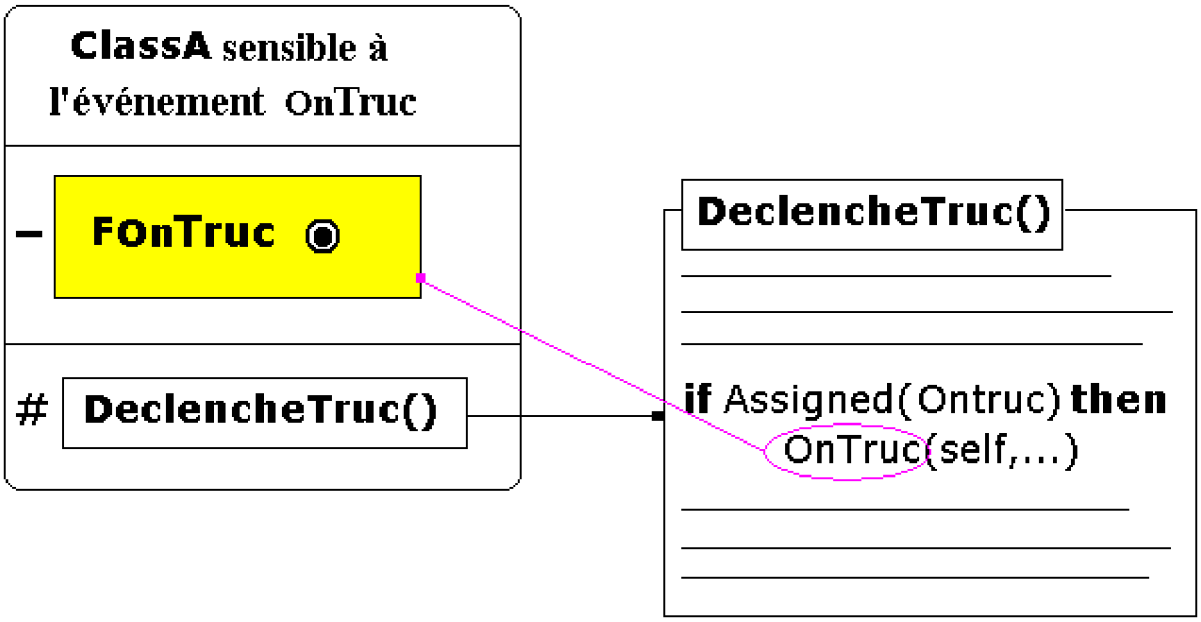


Version-1 du code source

```
Unit UDesignEvent ;

interface
 type
  EventTruc = procedure (Sender:TObject; info:string) of object ;

  ClassA = class
      private
```

```
        FOnTruc : EventTruc ;
    public
        OnTruc : EventTruc read FOnTruc  write FOnTruc ;
  end;

implementation


end.
```

- Il nous faut maintenant construire une méthode qui va déclencher l'événement, nous utilisons une procédure surchargeable dynamiquement afin de permettre des redéfinitions utlérieures par les descendants.



Lorsque l'événement se nomme OnXXX, les équipes de développement Borland donnent la fin du nom de l'événement OnXXX à la procédure redéfinissable. Ici pour l'événement **OnTruc** à la place de **DeclencheTruc**, nous la nommerons **Truc.**

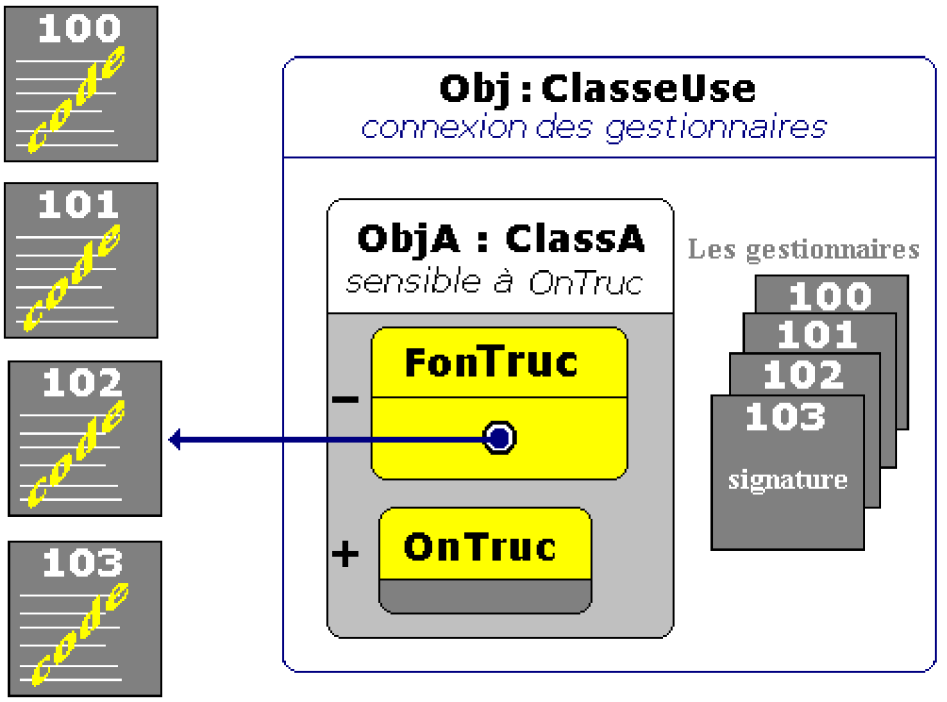
Version-2 du code source

```
Unit UDesignEvent ;

interface
 type
  EventTruc = procedure (Sender:TObject; info:string) of object ;
  ClassA = class
      private
          FOnTruc : EventTruc ;
      protected
          procedure Truc(s:string);virtual; // surchargeable dynamiquement
      public
          OnTruc : EventTruc read FOnTruc  write FOnTruc ;
   end;

implementation

 procedure ClassA.Truc(s:string);
```

```
begin
  if Assigned ( FOnTruc ) then
      FOnTruc (self , s)
end;

end.
```

- Pour terminer, il nous reste à définir un ou plusieurs gestionnaires possibles de l'événement OnTruc (ici nous en avons mis quatre), et à en connecter un à la propriété OnTruc de l'objet de classe ClassA.

Nous définissons une classe ClasseUse qui utilise sur un objet de classe ClassA l'événement OnTruc.



4 gestionnaires possibles pour le même événement

Version-3 du code source

```
Unit UDesignEvent ;

interface
 type   EventTruc = procedure (Sender:TObject; info:string) of object ;

  ClassA = class
      private
         FOnTruc : EventTruc ;
      protected
         procedure Truc(s:string);virtual; // surchargeable dynamiquement
      public
         ObjA : ClassA ;
         OnTruc : EventTruc read FOnTruc  write FOnTruc ;
         procedure LancerTruc; // Declenche l'événement OnTruc
```

```
  end;

  ClasseUse = class
      public
         procedure method_100(Sender:TObject; info:string);
         procedure method_101(Sender:TObject; info:string);
         procedure method_102(Sender:TObject; info:string);
         procedure method_103(Sender:TObject; info:string);
         procedure principale;
  end;

implementation

{------------------- ClassA  ------------------------}
 procedure ClassA.Truc(s:string);
 begin
   if Assigned ( FOnTruc ) then
      FOnTruc (self , s)
 end;

 procedure ClassA.LancerTruc ;
 begin
    ....
   Truc ("événement déclenché");
    ....
 end;
{------------------- ClasseUse  ------------------------}
 procedure ClasseUse.principale;
 begin
    //....
    ObjA := ClassA.Create ;
    ObjA.OnTruc := method_102 ; // connexion
    //....
    ObjA.LancerTruc ; // lancement
 end;

 procedure ClasseUse.method_100(Sender:TObject; info:string);
 begin
    //....
 end;

 procedure ClasseUse.method_101(Sender:TObject; info:string);
 begin
    //....
 end;

 procedure ClasseUse.method_102(Sender:TObject; info:string);
 begin
    //....
 end;

 procedure ClasseUse.method_103(Sender:TObject; info:string);
 begin
    //....
 end;

end.
```

Code pratique : une pile Lifo événementielle

Objectif : Nous livrons une classe de pile lifo héritant d'une Tlist (Un objet Tlist de Delphi, stocke un tableau de pointeurs, utilisé ici pour gérer une liste d'objets) et qui est réactive à l'empilement et au dépilement d'un objet. Nous suivons la démarche précédente en nous inspirant de son code final pour construire deux événements dans la pile lifo et lui permettre de réagir à ces deux événements.

```
unit  ULifoEvent ;

 interface
 uses  classes,Dialogs ;

 type
   DelegateLifo = procedure ( Sender: TObject ;  s :string ) of  object ;

  ClassLifo = class (TList)
  private
     FOnEmpiler : DelegateLifo ;
     FOnDepiler : DelegateLifo ;
  public
    function  Est_Vide  : boolean ;
    procedure  Empiler (elt : string ) ;
    procedure  Depiler ( var  elt : string ) ;
    property  OnEmpiler : DelegateLifo  read  FOnEmpiler  write  FOnEmpiler ;
    property  OnDepiler : DelegateLifo  read  FOnDepiler  write  FOnDepiler ;
end;

ClassUseLifo = class
 public
   procedure  EmpilerListener( Sender: TObject ;  s :string ) ;
   procedure  DepilerListener( Sender: TObject ;  s :string ) ;
   constructor  Create ;
   procedure  main ;
end;

implementation

 procedure  ClassLifo.Depiler( var  elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
    elt :=string (self.First) ;
    self.Delete(0) ;
    self.Pack ;
    self.Capacity := self.Count ;
    if  assigned(FOnDepiler)  then
      FOnDepiler ( self ,elt )
  end
 end;

 procedure  ClassLifo.Empiler(elt : string ) ;
 begin
  self.Insert(0 , PChar(elt)) ;
  if  assigned(FOnEmpiler)  then
     FOnEmpiler ( self ,elt )
 end;
```

Le type de l'événement : type pointeur de méthode (2 paramètres)

Champs privés stockant la valeur de l'événement (pointeur de méthode).

Evénement OnEmpiler :
- pointeur de méthode.

Evénement OnDepiler :
- pointeur de méthode.

self = la pile Lifo

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnDepiler **pointe vers elle, il est donc non nul.**
Sinon FOnDepiler **est nul (non assigné)**

L'instruction FOnDepiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnDepiler **pointe.**

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnEmpiler **pointe vers elle, il est donc non nul.**
Sinon FOnEmpiler **est nul (non assigné)**

L'instruction FOnEmpiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnEmpiler **pointe.**

```
function  ClassLifo.Est_Vide : boolean ;
begin
  result :=  self.Count = 0 ;
end;

 { ClassUseLifo }

constructor  ClassUseLifo.Create ;
begin
  inherited;
end;

procedure  ClassUseLifo.DepilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a depile : ' ,s) ;
end;

procedure  ClassUseLifo.EmpilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a empile : ' ,s) ;
end;

procedure  ClassUseLifo.main ;
var
  pileLifo : ClassLifo ;
  ch :string ;
 begin
   pileLifo := ClassLifo.Create ;
   pileLifo.OnEmpiler :=  EmpilerListener ;
   pileLifo.OnDepiler :=  DepilerListener ;
   pileLifo.Empiler( '[ eau ]' ) ;
   pileLifo.Empiler( '[ terre ]' ) ;
   pileLifo.Empiler( '[ mer ]' ) ;
   pileLifo.Empiler( '[ voiture ]' ) ;
   writeln ( 'Depilement de la pile :' ) ;
   while  not  pileLifo.Est_Vide  do
   begin
     pileLifo.Depiler(ch) ;
     writeln (ch) ;
   end;
   writeln ( 'Fin du depilement.' ) ;
   readln ;
 end;

end.
```

Classe de test créant une pile Lifo

Gestionnaire de l'événement OnDepiler : signature compatible avec DelegateLifo**.**

Gestionnaire de l'événement OnEmpiler : signature compatible avec DelegateLifo**.**

Méthode main **de test**

Instanciation d'une pile Lifo à tester.

Affectation de chaque gestionnaire à l'événement qu'il est chargé de gérer.

Empilement de 4 éléments

Dépilement de toute la pile

Application console **Project2.dpr** instanciant un objet de ClassUseLifo et lançant le test de la pile lifo par invocation de la méthode main.

```
program Project2;

{$APPTYPE CONSOLE}

uses   SysUtils , UlifoEvent ;
var execLifo : ClassUseLifo;
begin
  execLifo := ClassUseLifo.Create;
  execLifo.main
end.
```

exécution

```
C:\Program Files\Borland\Delphi7\Bin\Project
On a empile : [ eau ]
On a empile : [ terre ]
On a empile : [ mer ]
On a empile : [ voiture ]
Depilement de la pile :
On a depile : [ voiture ]
[ voiture ]
On a depile : [ mer ]
[ mer ]
On a depile : [ terre ]
[ terre ]
On a depile : [ eau ]
[ eau ]
Fin du depilement.
```

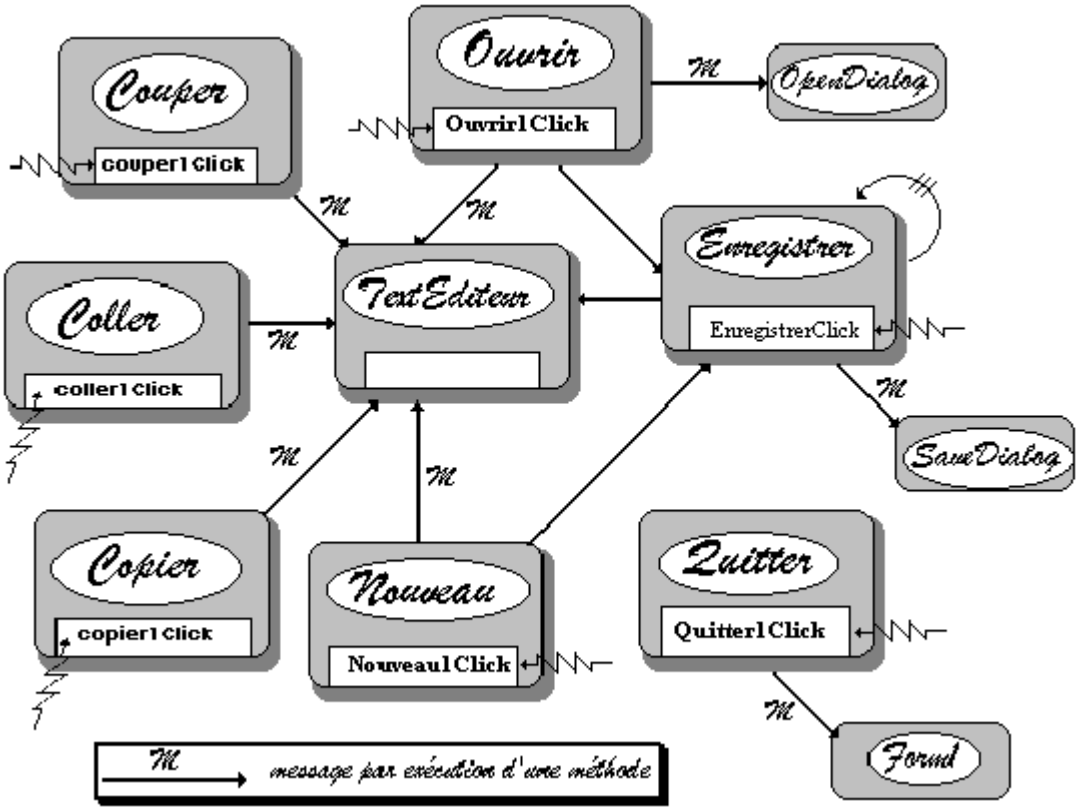
Exemple événementiel : un éditeur de texte

1. L'énoncé et les choix

Nous souhaitons développer rapidement un petit éditeur de texte qui nous permettra :

- de **lire** du texte à partir d'un fichier sur disque ou sur disquette (*ouvrir*),
- de **taper** du texte (*nouveau*),
- de **visualiser** le texte entré,
- d'effectuer sur ce texte les opérations classiques de *copier/ coller/ couper*,
- de le **sauvegarder** sur disque ou sur disquette (*enregistrer*).

Les objets potentiels réagiront à ces événements selon le graphe ci-dessous



L'objet ***TextEditeur*** est l'objet central sur lequel agissent tous les autres objets, il contiendra le texte à éditer.
En faisant ressortir les opérations à effectuer, l'utilisation de la notion d'abstraction est naturelle.

Nous envisageons les actions suivantes obtenues par réaction à un événement Click de souris (choix des objets du graphe précédent):

nouveau *(utilise un objet à définir)*
couper *(utilise un objet à définir)*

copier *(utilise un objet à définir)*
coller *(utilise un objet à définir)*
ouvrir *(utilise un objet de dialogue)*
enregistrer *(utilise un objet de dialogue)*
quitter *(utilise un objet prédéfini Form)*

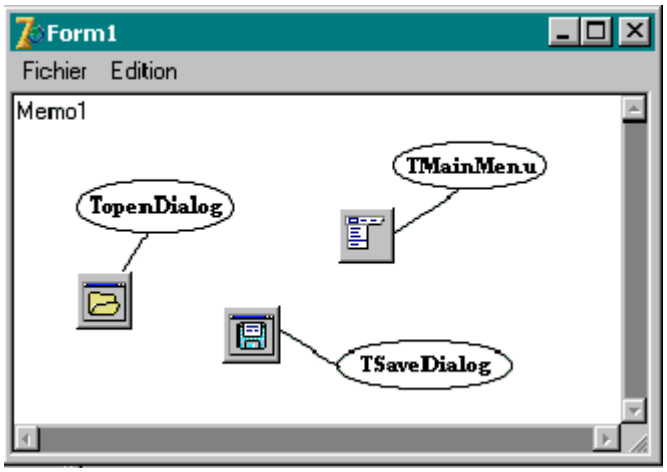
2. Les objets de composants

Delphi étant orienté objet nous allons exploiter complètement l'analyse présentée dans le graphe événementiel ci-haut en mettant en place quatre objets du genre composants visuels ou non visuels de Delphi.

L'interface choisie

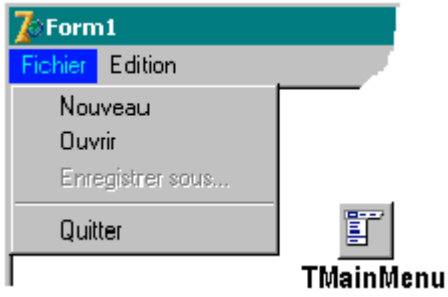
*Une fiche (**Form1**) comportant :*
- Un Tmemo avec deux ascenseurs

Trois composants non visuels :
- **TMainmenu** (une barre de 2 menus)
- **TopenDialog** (pour le chargement de fichier)
- **TSaveDialog** (pour la sauvegarde d'un texte)



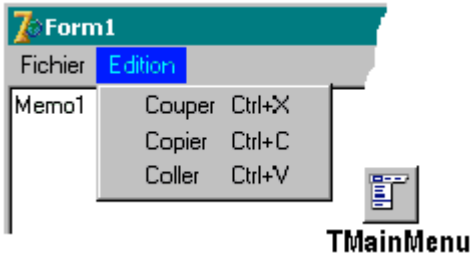
Le composant TMainmenu(1er menu)

comporte un premier menu *Fichier à 5 items*

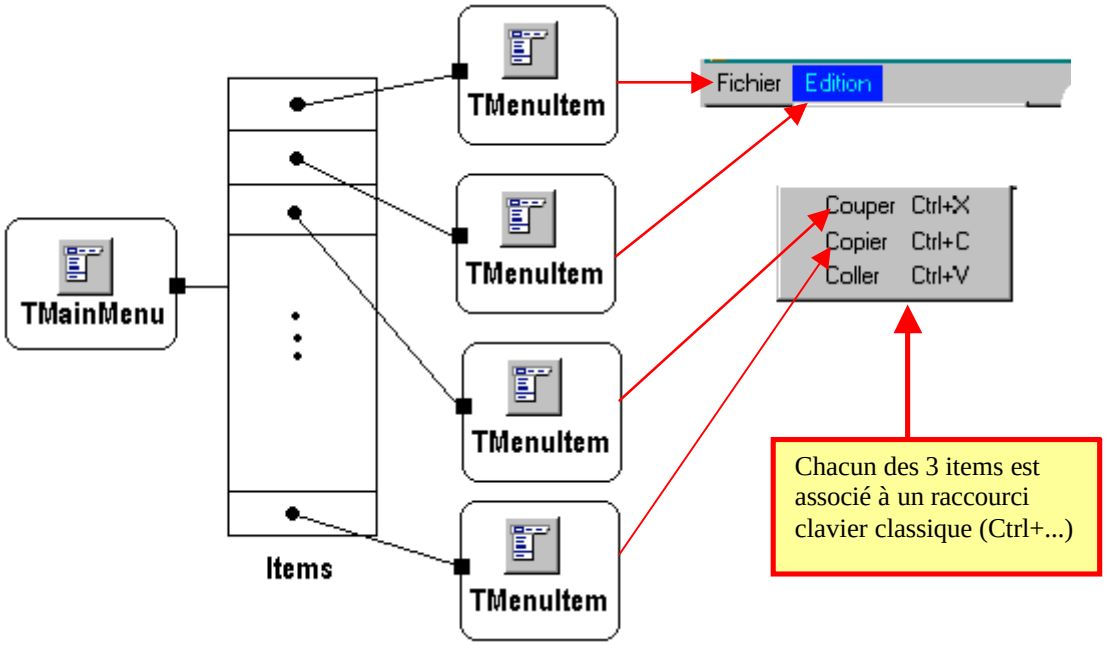


Le composant TMainmenu (2ème menu)

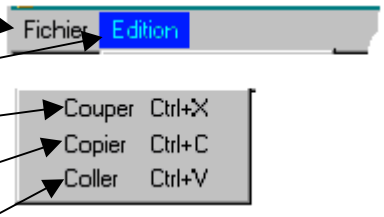
comporte un deuxième menu *Edition à 3 items.*



Les menus dans la barre et les sous-menus sont tous des objets de classe TmenuItem qui sont gérés à travers le champ Items d'un objet de classe TMainMenu :




```
object MainMenu1: TMainMenu
  ....
  object fichier1: TMenuItem
    Caption = 'fichier'
  end
  object edition1: TMenuItem
    Caption = 'edition'
    object couper1: TMenuItem
      Caption = 'couper'
    end
    object copier1: TMenuItem
      Caption = 'copier'
    end
    object coller1: TMenuItem
      Caption = 'coller'
    end
  end
end .....
```



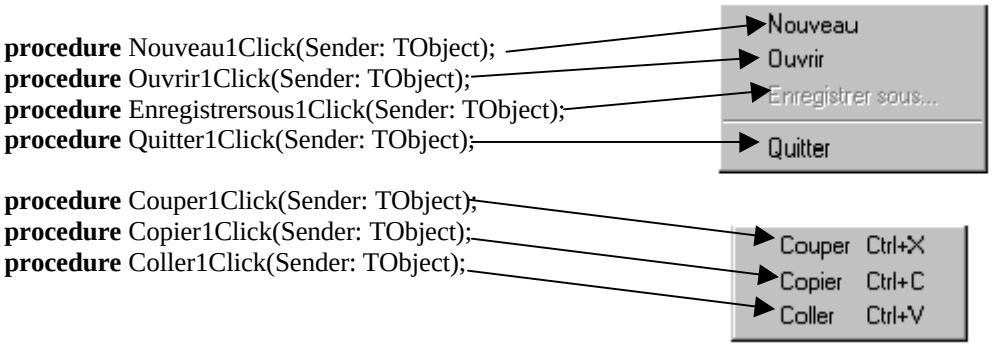
Correspondance entre le code caché par Delphi et l'inclusion des objets de TmenuItem dans un TmainMenu.

Mise en place de la gestion des événements.

A chacun des items utilisables de chacun des 2 menus, il nous faut associer un gestionnaire d'événement qui indique au programme comment il doit réagir lorsque l'utilisateur sélectionne un champ de l'un des menus par un click de souris. Nous avons vu que Delphi contient le mécanisme d'association des événements à nos gestionnaires. Beaucoup de contrôles (classes visuelles)et beaucoup d'autres classes non visuelles comme les TMenuItem, de Delphi sont sensibles à l'événement de click de souris : **property** OnClick : TnotifyEvent.

Les gestionnaires de l'événement OnClick pour les TMenuItem

Voici les en-têtes des 7 gestionnaires de OnClick automatiquement construits:



Voici pour chaque gestionnaire le code écrit par le programmeur.

Le code du gestionnaire *Quitter1Click*

```
procedure TForm1.Quitter1Click(Sender: TObject);
begin
  Close; //écrit par le développeur (fermeture de la fenêtre)
end;
```

Le code du gestionnaire *Ouvrir1Click*

```
procedure TForm1.Ouvrir1Click(Sender: TObject);
begin
  if  OpenDialog1.Execute then //écrit par le développeur
  begin
    Enregistrersous1.enabled:=true;
    TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
  end
end;
```

Le code du gestionnaire *Enregistrersous1Click*

```
procedure TForm1.Enregistrersous1Click(Sender: TObject);
begin
  if  SaveDialog1.Execute then // écrit par le développeur
  begin
    Enregistrersous1.enabled:=false;
    TextEditeur.Lines.SaveToFile( SaveDialog1.FileName );
  end
end;
```

Le code du gestionnaire *Couper1Click*

```
procedure TForm1.Couper1Click(Sender: TObject);
begin
  TextEditeur.CutToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Copier1Click*

```
procedure TForm1.Copier1Click(Sender: TObject);
begin
  TextEditeur.CopyToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Coller1Click*

```
procedure TForm1.Coller1Click(Sender: TObject);
begin
  TextEditeur.PasteFromClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Nouveau1Click*

```
procedure TForm1.Nouveau1Click(Sender: TObject);
begin
  TextEditeur.Clear; &not; écrit par le développeur
  Enregistrersous1.enabled:=true &not; écrit par le développeur
end;
```

Il est à noter que nous avons écrit au total 16 lignes de programme Delphi pour construire notre micro-éditeur. Ceci est rendu possible par la réutilisabilité de méthodes déjà intégrées dans les objets de Delphi et en fait présentes dans le système Windows.

Vous pouvez voir la puissance de ce **RAD** lorsque vous observez par exemple l'instruction qui a permis de faire exécuter le "copier" ou le "chargement d'un fichier" :

```
TextEditeur.CopyToClipboard;
```

La méthode **CopyToClipboard** s'applique à l'objet **TextEditeur** qui est de la classe TMemo; cette instruction correspond à un ensemble complexe d'opérations de bas niveau :
le TMemo autorise une suite complexe d'opérations permettant de sélectionner un texte écrit sur plusieurs lignes du TMemo. Il les affiche en surbrillance et la méthode **CopyToClipboard** récupère le texte sélectionné puis le recopie enfin dans le presse-papier du système.

```
TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
```

Nous n'avons par ailleurs eu aucun code particulier à écrire pour le chargement de fichier texte, la méthode **LoadFromFile** présente dans l'objet **Lines** (de classe TStrings) effectue toutes les actions.

A titre de travail personnel il est recommandé d'enrichir ce micro-éditeur avec la possibilité de changer la police de caractère, la couleur du fond etc…

5.6 Programmation défensive
(les exceptions)

Plan du chapitre:

Introduction

1.Notions de défense et de protection

1.1 Outils participant à la programmation défensive
1.2 Rôle et mode d'action d'une exception
1.3 Gestion de la protection du code
Fonctionnement sans incident
Fonctionnement avec incident
1.4 Effets dus à la position du bloc except...end
Fonctionnement sans incident
Fonctionnement avec incident
1.5 Interception d'une exception d'une classe donnée
1.6 Ordre dans l'interception d'une exception
Interception dans l'ordre de la hiérarchie
Interception dans l'ordre inverse

2. Traitement d'un exemple de protections

2.1 Le code de départ de l'unité
2.2 Code de la version.1 (premier niveau de sécurité)
2.3 Code de la version.2 (deuxième niveau de sécurité)

Introduction

Nous allons montrer comment on peut concevoir la programmation défensive en protégeant directement le code à l'aide de la notion d'exception (semblable à celle du C++ ou d'Ada). L'objectif principal est d'améliorer la qualité de " *robustesse* " (définie par B.Meyer) d'un logiciel. L'utilisation des exceptions avec leur mécanisme intégré, autorise la construction rapide et néanmoins efficace de logiciels robustes.

Rappelons au lecteur que la sécurité d'une application est susceptible d'être mise à mal par toute une série de facteurs :

- les problèmes liés au matériel : par exemple la perte subite d'une connection à un port, un disque défectueux...
- les actions imprévues de l'utilisateur, entrainant par exemple une division par zéro...

1. Notions de défense et de protection

A l'occasion de la traduction Algorithme à langage évolué comme pascal, nous avons répertorié en plus des actions algorithmiques, **des actions de sécurité** et des actions ergonomiques qui doivent elles aussi être programmées.

La partie ergonomie est incluse dans la notice consacrée aux interfaces de communication logiciel-utilisateur.
La partie sécurité a déjà été abordée ailleurs, nous regroupons ici ce que nous devons connaître.

Pour obtenir une certaine " **robustesse** " dans nos programmes nous savons déjà que la sécurité doit porter au moins sur :

- les domaines de définitions des données,
- les contraintes d'implantation,
- le filtrage des saisies,
- les problèmes de transtypage

Toutefois les faiblesses dans un logiciel pendant son exécution, peuvent survenir : lors des entrées-sorties, lors de calculs mathématiques interdits (comme la division par zéro), lors de fausses manoeuvres de la part de l'utilisateur, ou encore lorsque la connexion à un périphérique est inopinément interrompue.

La ***programmation défensive*** est plus une attitude de pensée et de comportement qu'une nouvelle méthode. Cette attitude consiste à prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes et donc à prévoir une réponse adaptée à chaque type de situation.

La comparaison des coûts de différents ratios de productivité établie par B.Boehm, montre que l'exigence de fiabilité est l'une des plus chères (surcoût de 87%). Cette remarque pourrait en apparence nous conduire à croire que ce facteur est donc une affaire de spécialistes et ne constitue pas une préoccupation dans un discours d'initiation. Nous allons voir qu'il n'en est rien et qu'en fait notre méthode de travail et les outils que nous utilisons concourent à une attitude de programmation défensive sans que nous contraignions fortement notre pensée en ce sens.

Nous pensons enfin qu'il est bon d'induire dans l'esprit du développeur en programmation visuelle débutant des idées fondées sur des pratiques méthodiques qui lui éviteront l'écueil de " l'art indiscipliné " du logiciel, mais qui pourraient lui donner le goût de continuer dans cette discipline.

1.1 Outils participant à la programmation défensive

Voici cinq secteurs d'activité parmi ceux que nous connaissons, participant à une méthode de programmation défensive.

La modularité

Le principe du découpage en modules restreint la propagation des effets perturbateurs sur d'autres modules à partir d'une erreur survenant dans un module donné.

L'encapsulation

Associée à la modularité, elle protège les constituants internes d'un module (champs et méthodes).

Les TAD (types abstraits)

En fournissant un outil de spécification des données avec des préconditions sur la validité des opérateurs, un TAD assure la description des vérifications de domaines.

L'utilisation de méthodes systématiques

Comme l'algorithmique, la programmation par la syntaxe, les machines abstraites, les générateurs d'automates,... ces outils encouragent l'étudiant à s'essayer à une programmation sûre.

L'utilisation d'un RAD visuel comme Delphi

Ce genre de système de développement apporte un certains nombres d'avantages en la matière :

- il autorise la mise en œuvre des cinq secteurs précédents d'activité,
- il fournit au programmeur des kits d'objets sécurisés et réutilisables,
- il permet de construire rapidement des interfaces qui participent à une meilleure défense du logiciel contre des actions non valides (par exemple en utilisant la méthode de séquencement par plans d'action).

A côté de cet éventail d'outils intégrant en général la notion de programmation défensive, il existe un outil spécifique dédié à ce genre de programmation : ***les exceptions***.

1.2 Rôle et mode d'action d'une exception

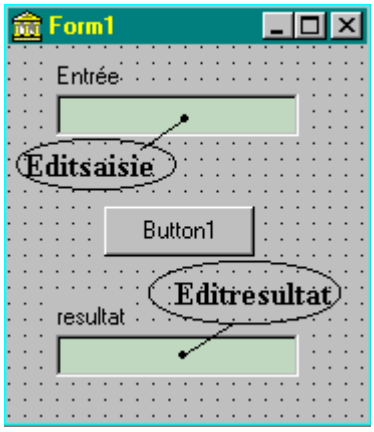
Rôle d'une exception

Réservé jusqu'à présent aux spécialistes en Ada, en C++ ou en Java, cet outil est mis dans le RAD Delphi à la portée du débutant. Comme son nom l'indique, une exception est chargée de signaler un comportement *exceptionnel* (mais prévu) d'une partie spécifique d'un logiciel. Dans les langages de programmation actuels, les exceptions font partie du langage lui-même. C'est le cas de Delphi qui intègre les exceptions comme une classe particulière : la classe **Exception**. Cette classe contient un nombre important de classes dérivées.

Comment agit une exception

Dès qu'une erreur se produit comme un manque de mémoire , un calcul impossible, un fichier inexistant, un transtypage non valide,..., un objet de la classe adéquate dérivée de la classe **Exception** est instancié. Nous dirons que le logiciel " *déclenche une exception* ".

Exemple : soit une saisie d'un entier dans un Tedit nommé Editsaisie.



Objets visuels :

Editsaisie: TEdit;
Editresultat: TEdit;
Button1: TButton;

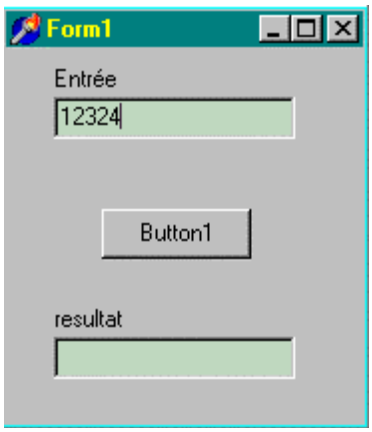
Actions :

L'utilisateur entre un entier dans Editsaisie, il clique sur le bouton Button1 qui effectue un transtypage dans une variable locale " n : integer " puis il se désactive et le Tedit Editresultat, change de couleur.

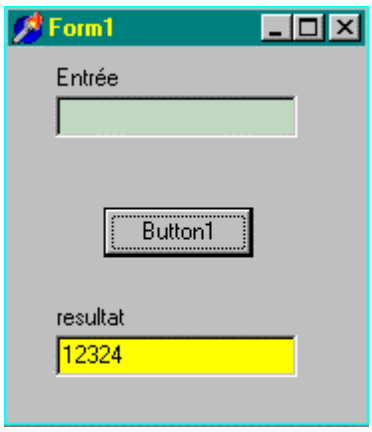
Le gestionnaire d'événement clic de Button1 est alors le suivant :

```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
  Editresultat.color:=clyellow;
  Editresultat.text:= Editsaisie.text;
  Editsaisie.clear ;
end;
```

Une exécution sans incident donne ceci :



avant clic sur Button1



après clic sur Button1

Lors de l'exécution, supposons que nous entrons dans Editsaisie la chaîne " asa12324 " qui n'est pas un entier.
La fonction **StrtoInt** déclenche une exception et nous envoie un message d'erreur général sur l'incident qui vient de se produire : (après clic sur Button1 Message d'erreur)

En outre, l'incident a arrêté l'exécution du code dans le gestionnaire Button1Clic. Le code a été exécuté normalement jusqu'à l'instruction de transtypage qui a déclenché l'exception. Le reste des lignes de code a été ignoré :

```
procedure TForm1.Button1Click(Sender: TObject);
var n:integer;
begin
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
  Editresultat.color:=clyellow;
  Editresultat.text:= Editsaisie.text;
  Editsaisie.clear ;
end;
```

cette partie n'a pas été exécutée

Si nous voulons que le message soit plus explicite, ou si nous voulons par exemple que malgré tout une certaine partie du code s'exécute en fournissant des valeurs par défaut malgré l'incident, nous devons gérer nous-mêmes cette exception.
Afin de pouvoir gérer une exception déclenchée, il nous faut disposer d'un gestionnaire d'exception qui permette de traiter tous les types d'exception.

1.3 Gestion de la protection du code

Le langage Delphi contient un mécanisme appelé gestionnaire d'exception qui s'insère dans les lignes de code là où nous souhaitons assurer une protection.

Syntaxe du gestionnaire : mots clefs (try, except)

```
try
    - ...
    <lignes de code à protéger>
    - ...
except
    - ...
    <lignes de code réagissant à l'exception>
    - ...
end ;
```

Principe de fonctionnement d'un tel gestionnaire :

Dès qu'une exception est déclenchée dans le bloc de lignes compris entre **try**....**except**, il y a déroutement de l'exécution (arrêt d'exécution séquentielle du code) vers la première ligne du bloc **except**...**end** et l'exécution continue séquentiellement à partir de cet endroit.
Reprenons l'exemple précédent légèrement modifié dans le code du gestionnaire du clic de Button1.

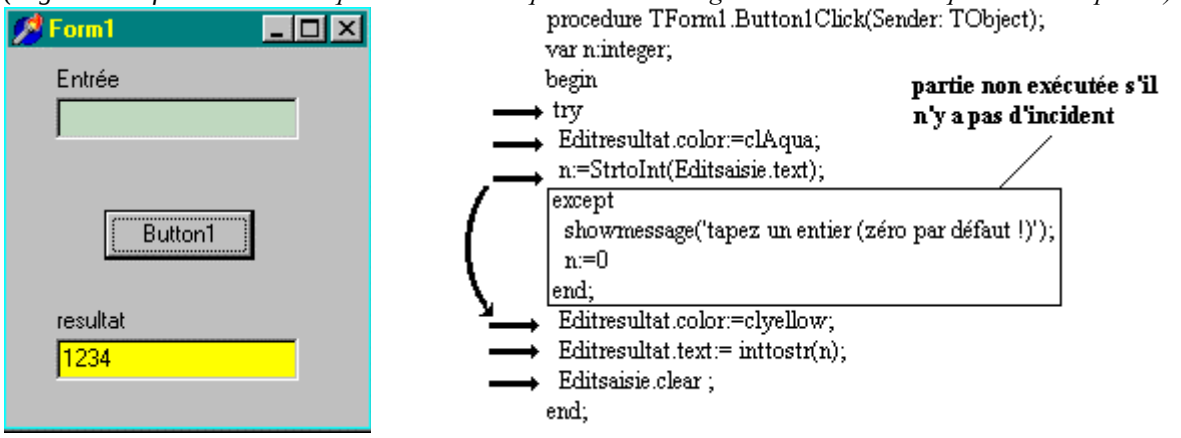
```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
  Editresultat.color:=clyellow;
  Editresultat.text:= inttostr(n);
  Editsaisie.clear ;
end;
```

Mettons en place une protection de l'instruction incriminée, tout en conservant l'exécution des lignes de code suivantes. Comme la variable " n " n'aura pas de valeur à cause de l'incident, nous lui attribuons la valeur zéro par défaut si un incident se produit.

```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
 try
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
 except
  showmessage('tapez un entier (zéro par défaut !');
  n:=0
 end;
 Editresultat.color:=clyellow;
 Editresultat.text:= inttostr(n);
 Editsaisie.clear ;
end;
```

1.3.1 - Fonctionnement sans incident :

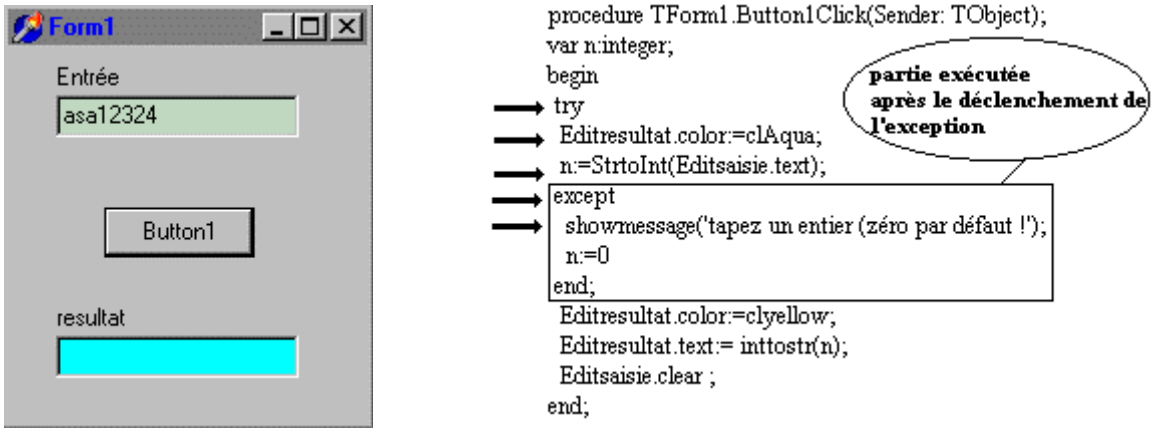
(Légende : la flèche ➡ indique l'exécution séquentielle de la ligne de code devant laquelle elle est placée).

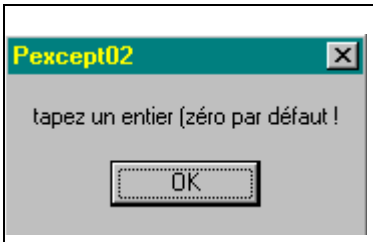


Les lignes de code sont exécutées séquentiellement sauf le bloc **except**...**end** (qui n'est exécuté qu'en cas d'incident).

1.3.2 - Fonctionnement avec incident :

Nous entrons dans Editsaisie comme précédemment une valeur non entière.

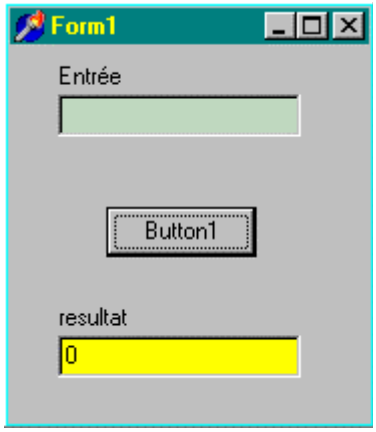




Message envoyé par la fonction Showmessage dans le bloc **except**...**end**.

Ensuite le code continue à s'exécuter en séquence à partir de n :=0 ;...

*Suite du déroulement séquentiel de l'exécution dans le bloc **except…end** :*




```
procedure TForm1.Button1Click(Sender: TObject);
var n:integer;
begin
 try
  Editresultat.color:=clAqua;
  n:=StrtoInt(Editsaisie.text);
 except
  showmessage('tapez un entier (zéro par défaut !');
  n:=0
 end;
  Editresultat.color:=clyellow;
  Editresultat.text:= inttostr(n);
  Editsaisie.clear ;
end;
```

partie exécutée après le déclenchement de l'exception

*En fait dans cet exemple, n'importe quelle exception déclenchée dans le bloc **try…except** déroute vers le bloc **except…end**.*

1.4 Effets dus à la position du bloc except...end

Afin de bien comprendre cette notion de déroutement (dont le placement est à la charge du programmeur) observons ce qu'apporte une modification de la place du bloc **except...end** au déroulement du code pendant l'exécution.
Soit, dans le même exemple, le nouveau code dans Button1Clmick où nous avons repoussé le bloc **except**...**end** à la fin.

```
procedure TForm1.Button1Clic(Sender: TObject);
var n:integer;
begin
 try
   Editresultat.color:=clAqua;
   n:=StrtoInt(Editsaisie.text);
   Editresultat.color:=clyellow;
   Editresultat.text:= inttostr(n);
   Editsaisie.clear ;
 except
   showmessage('tapez un entier (zéro par défaut !');
   n:=0
 end;
end;
```

Puis exécutons le programme.

1.4.1 - Fonctionnement sans incident :

Identique au précédent paragraphe, le bloc **except...end** étant ignoré.

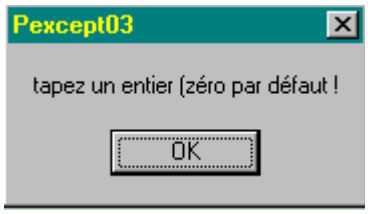
1.4.2 - Fonctionnement avec incident :

(Légende : la flèche ➡ indique l'exécution séquentielle de la ligne de code devant laquelle elle est placée).

```
procedure TForm1.Button1Click(Sender: TObject);
var n:integer;
➡ begin
➡ try
➡  Editresultat.color:=clAqua;
➡  n:=StrtoInt(Editsaisie.text);
   Editresultat.color:=clyellow;
   Editresultat.text:= inttostr(n);
   Editsaisie.clear ;
➡ except
➡  showmessage('tapez un entier (zéro par défaut !');
➡  n:=0
➡ end;
end;
```

partie non exécutée suite à la levée de l'exception

le code continue son déroulement dans le bloc **except...end** en " sautant " trois instructions.




```
➡ except
➡  showmessage('tapez un entier (zéro par défaut !');
➡  n:=0
➡ end;
end;
```

Etat final des résultats après traitement de l'exception :

les deux Tedit Editresultat et Editsaisie n'ont pas changé puisque les instructions:

Editresultat.color:=clyellow;
Editresultat.text:= inttostr(n);
Editsaisie.clear ;

n'ont pas été exécutées, par suite du déroutement du code.

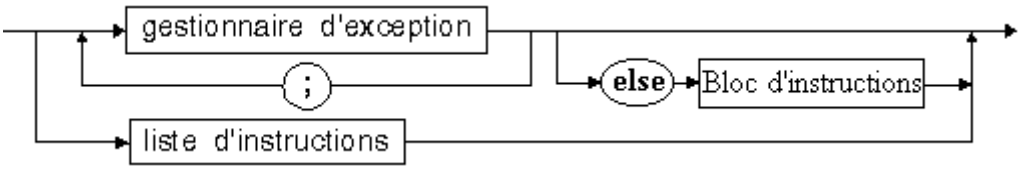
Nous notons que cette méthode de déroutement du code est très proche du fonctionnement d'une machine de Von Neumann. Nous venons de voir comment intercepter une exception quelconque sans savoir exactement sa catégorie. Nous pouvons en fait intercepter une exception d'une manière encore plus " fine " avec le gestionnaire **try...except...end**.
Il nous permet de sélectionner la classe exacte de l'exception et de ne faire fonctionner le déroutement du code que pour une exception définie à l'avance.

1.5 Interception d'une exception d'une classe donnée

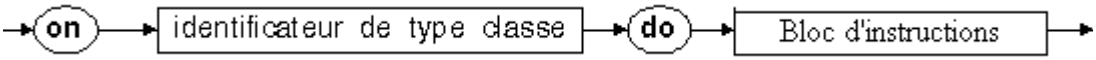
Diagrammes de syntaxe du gestionnaire :

<instruction try> :

<Bloc d'exception> :

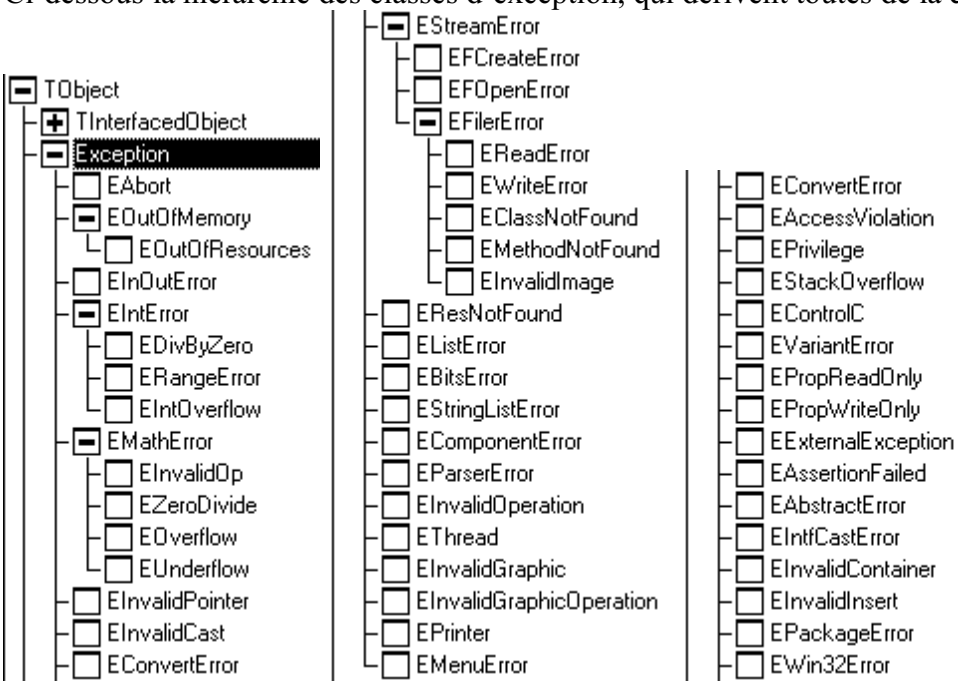


<gestionnaire d'exception> :



Différentes classes d'exception Delphi

Ci-dessous la hiérarchie des classes d'exception, qui dérivent toutes de la classe **Exception**:



Exemple d'écriture d'un gestionnaire :

```
try
        <lignes de code à protéger>
    except
     On EConvertError do begin
        <lignes de code réagissant à l'exception EConvertError >
```

```
        end ;
        On EintError do begin
            <lignes de code réagissant à l'exception EintError >
        end ;
        else begin
            <lignes de code réagissant à d'autres exceptions >
        end ;
       end ;
```

Principe de fonctionnement d'un tel gestionnaire

Dès qu'une exception est déclenchée dans le bloc de lignes compris entre **try**....**except**, il y a déroutement de l'exécution (arrêt d'exécution séquentielle du code) vers le bloc **except**...**end**.
Le sélecteur de gestionnaire d'exception **on...do** fonctionne approximativement comme un **case...of**, en n'exécutant que le champ sélectionné.
Supposons que l'exception levée soit de la classe EintError ; l'instruction try n'exécute alors que le code du " bon " gestionnaire en l'occurrence :

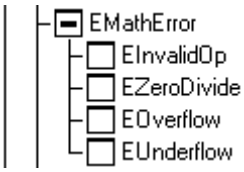
```
  On EintError do begin
    <lignes de code réagissant à l'exception EintError >
  end ;
```

Puis l'exécution se poursuit après le **end** du bloc **except...end**.

1.6 Ordre dans l'interception d'une exception

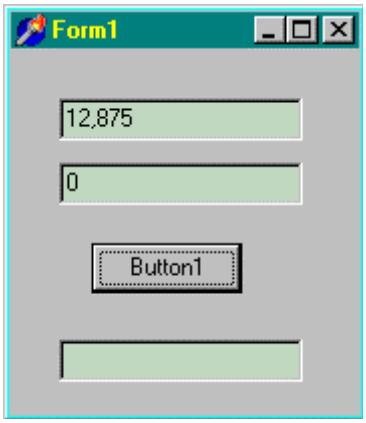
A la différence d'un "**case** … **of**" pascal, le choix du sélecteur de gestionnaire (**on**...**do**) s'effectue séquentiellement dans l'ordre d'écriture des lignes de code. On choisira donc, lorsqu'il y a une hiérarchie entre les exceptions à intercepter, de placer le code de leurs gestionnaires dans l'ordre inverse de la hiérarchie.

Exemple : division par zéro dans un calcul en virgule flottante
EMathError est la classe des exceptions pour les erreurs de calcul.



EZeroDivide indique la division par zéro dans une telle opération.

Programmons un calcul à partir d'un bouton :



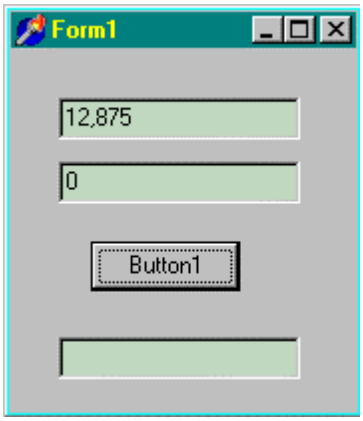

```
procedure TForm1.Button1Clic(Sender:
TObject);
var x,y,z:real;
begin
 try
   x:=StrtoFloat(Edit1.text);
   y:=StrtoFloat(Edit2.text);
   z:=x /y ;
   Edit3.text:=FloattoStr(z);
 except
  ............
  ............
 end;
end;
```

Au départ nous avons edit1.text = 12,875 et edit2.text = 0. Le calcul est erroné car x vaut 12,85 et y vaut zéro ce qui va induire une erreur de division par zéro dans l'instruction z := x / y .

Observons ce qui se passe lorsque nous interceptons les exceptions **EMathError** et **EZeroDivide**.

1.6.1 - Interception dans l'ordre de la hiérarchie :

La sélection d'exception est programmée dans l'ordre de la hiérarchie des classes :
EMathError
|____ **EZeroDivide**

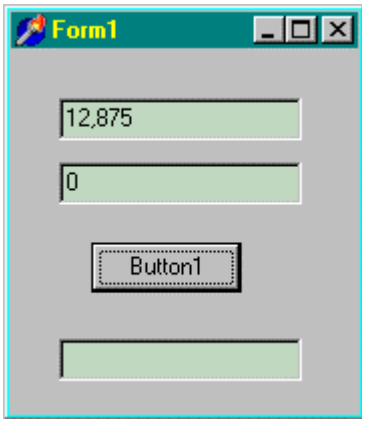



```
procedure TForm1.Button1Clic(Sender: TObject);
var x,y,z:real;
begin
 try
   x:=StrtoFloat(Edit1.text);
   y:=StrtoFloat(Edit2.text);
   z:=x /y ;
   Edit3.text:=FloattoStr(z);
 except
   on EMathError do
     Edit3.text:='Erreur générale';
   on EZeroDivide do
     Edit3.text:='division par zéro';
 end;
end;
```

EMathError est interceptée en premier.

1.6.2 - Interception dans l'ordre inverse :

La sélection d'exception est programmée dans l'ordre inverse de la hiérarchie des classes.



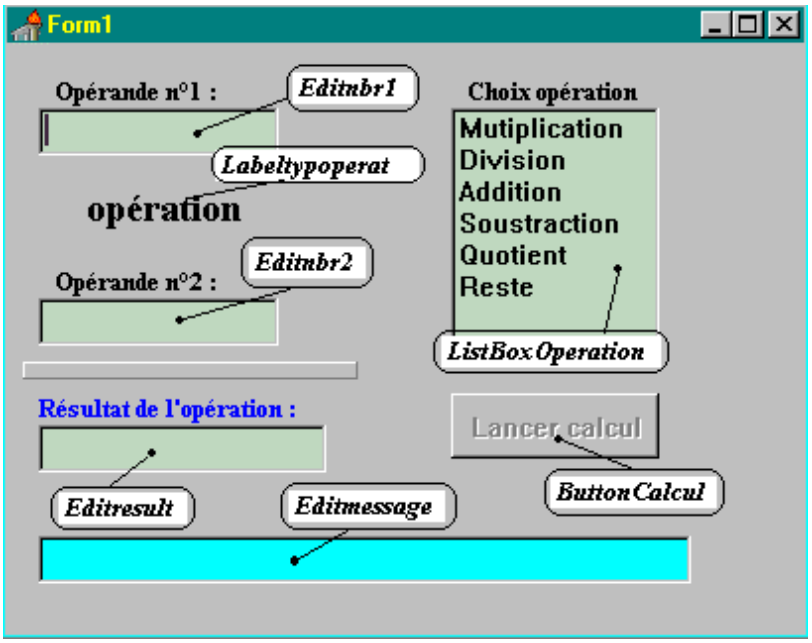

```
procedure TForm1.Button1Clic(Sender: TObject);
var x,y,z:real;
begin
 try
   x:=StrtoFloat(Edit1.text);
   y:=StrtoFloat(Edit2.text);
   z:=x /y ;
   Edit3.text:=FloattoStr(z);
 except
  on EZeroDivide do
     Edit3.text:='division par zéro';
   on EMathError do
     Edit3.text:='Erreur générale';
 end;
end;
```

C'est **EZeroDivide** qui est interceptée en premier.

2. Traitement d'un exemple de protections

Reprenons comme base le premier exemple étudié dans le chapitre sur les interfaces.
Supposons que nous voulons élargir notre interface de calcul aux opérations entières : Mutiplication, Division, Addition, Soustraction, Quotient, Reste.
Nous limiterons nos entiers au type **Smallint** Delphi identique au type integer du pascal (Smallint = -32768..32767, signé sur 16 bits).

Interface retenue



Editnbr1: TEdit;

Editnbr2: TEdit;

Editmessage: TEdit;

Editresult: TEdit;

ListBoxOperation: TListBox;

ButtonCalcul: TBitBtn;

Labeltypoperat: TLabel;

2.1 Le code de départ de l'unité

```
Champ privé de la classe Tform1 :
  Toper:array[0..5]of string;
```

implementation

Le tableau Toper contient les symboles des opérateurs entiers, il est initialisé lors de l'activation de la fiche

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Toper[0]:='*'; Toper[1]:='/';
  Toper[2]:='+'; Toper[3]:='-';
  Toper[4]:=' div '; Toper[5]:=' mod ';
end;
```

L'utilisateur choisit par sélection dans le ListBoxOperation l'opération qu'il veut effectuer ; l'étiquette Labeltypoperat affiche le symbole associé :

```
procedure TForm1.ListBoxOperationClic(Sender: TObject);
begin
  Labeltypoperat.caption:=Toper[ListBoxOperation.ItemIndex];
end;
```

Les deux Edit de saisie Editnbr1 et Editnbr2 sont sensibles à l'événement OnChange qui permet de déverrouiller le bouton de calcul :

```
procedure TForm1.Editnbr1Change ( Sender :
TObject);
begin
  ButtonCalcul.enabled:=true;
end;
```

```
procedure TForm1.Editnbr2Change  ( Sender :
TObject);
begin
  ButtonCalcul.enabled:=true;
end;
```

Une fois que les entiers sont entrés et le genre d'opération sélectionné, le ButtonCalcul lance le calcul sur un clic de l'utilisateur :

```
procedure TForm1.ButtonCalculClic(Sender: TObject);
var op1,op2,result:smallint;
begin
  ButtonCalcul.enabled:=false;
  op1:=strtoint(Editnbr1.text);
  op2:=strtoint(Editnbr2.text);
  case ListBoxOperation.ItemIndex of
    0: result:=op1 * op2;
    1: result:=trunc(op1 / op2);
    2: result:=op1 + op2;
    3: result:=op1 - op2;
    4: result:=op1 div op2;
    5: result:=op1 mod op2;
  end;
  Editresult.text:=inttostr(result);
```

```
  Editmessage.text:=inttostr(op1)+Toper[ListBoxOperation.ItemIndex]
      +' '+inttostr(op2)+'= '+Editresult.text;
end;
```

A ce stade la saisie n'est pas encore sécurisée. Afin que le lecteur puisse retrouver les éléments déjà traités dans le chapitre sur les interfaces, nous reprenons comme premier niveau de sécurité le même genre de programmation : synchronisation des 3 objets de saisie Editnbr1, Editnbr2 et ListBoxOperation, puis programmation par plans d'action.

2.2 Code de la version.1 (premier niveau de sécurité)

Champs privés de la classe Tform1**:**
oper1,oper2,operat:boolean; *//les 3 drapeaux*
Toper:**array**[0..5]**of** string;

implementation

- Le tableau Toper contient les symboles des opérateurs entiers, il est initialisé lors de l'activation de la fiche (pas d'ajout de code ni de changement).
- Ajout de la méthode **RAZtout** positionnant l'interface à son état initial (correspondant à la table des états initiaux) :

```
procedure TForm1.RAZTout;
begin
  ButtonCalcul.enabled:=false;
  Editnbr1.clear;
  Editnbr2.clear;
  Editresult.clear;
  Editmessage.clear;
  oper1:=false;
  oper2:=false;
  operat:=false;
end;
```

Adjonction de la méthode TestEntrees chargée de lancer l'activation deButtonCalcul si les trois drapeaux sont tous levés :

```
procedure TForm1.TestEntrees;
begin
  if oper1 and oper2 and operat then
     ButtonCalcul.enabled:=true
end;
```

L'utilisateur choisit par sélection dans le ListBoxOperation l'opération qu'il veut effectuer, l'étiquette Labeltypoperat affiche le symbole associé. Adjonction dans le code du drapeau et du test :

```
procedure TForm1.ListBoxOperationClic(Sender: TObject);
begin
  operat:=true;
  TestEntrees;
  Labeltypoperat.caption:=Toper[ListBoxOperation.ItemIndex];
end;
```

Les deux Edit de saisie Editnbr1 et Editnbr2 sont sensibles à l'événement OnChange ; voici l'ajout du code de plans d'action dans les gestionnaires de cet événement :

```
procedure TForm1.Editnbr1Change(Sender: TObject);
begin
 if Editnbr1.text<>'' then begin
   oper1:=true;
   TestEntrees;
 end
 else begin
   ButtonCalcul.enabled:=false;
   oper1:=false; // drapeau de Editnbr1 baissé
 end
end;
```

```
procedure TForm1.Editnbr12Change(Sender: TObject);
begin
 if Editnbr2.text<>'' then begin
    oper2:=true;
    TestEntrees;
 end
 else begin
    ButtonCalcul.enabled:=false;
    oper2:=false; // drapeau de Editnbr2 baissé
 end
end;
```

Le code du plan d'action associé au ButtonCalcul reste strictement le même.

Nous proposons dans le paragraphe qui suit, un complément de sécurité apporté par des interceptions d'exception.

2.3 Code de la version.2 (deuxième niveau de sécurité)

Tout le code de la version.1 reste identique dans la version.2, les adjonctions sont uniquement dans le gestionnaire du ButtonCalcul. Nous lançons les levées d'exception lorsque les incidents ont lieu. Nous avons ajouté une méthode signe qui renvoie +1 ou -1 selon le signe de l'entier d'entrée et d'une constante d'entier maximum:

```
const Maxint=32767;
function signe(n:smallint):smallint;
begin
  if n>0 then signe:=1
  else if n<0 then signe:=-1
        else signe:=0
end;
```

Le code est protégé par les exceptions suivantes :

```
procedure TForm1.ButtonCalculClic(Sender: TObject);
var op1,op2,result:smallint;
begin
 ButtonCalcul.enabled:=false;
try
  op1:=strtoint(Editnbr1.text);
 except
  on ERangeError do
   if Editnbr1.text[1]='-' then
      op1:=-Maxint
   else op1:=Maxint;
  on EConvertError do op1:=Maxint;
end;
try
 op2:=strtoint(Editnbr2.text);
except
 on EConvertError do op2:=Maxint;
 on ERangeError do
  if Editnbr2.text[1]='-' then
    op2:=-Maxint
  else op2:=Maxint;
end;

try
 case ListBoxOperation.ItemIndex of
  0: result:=op1 * op2;
  1: result:=trunc(op1 / op2);
  2: result:=op1 + op2;
  3: result:=op1 - op2;
  4: result:=op1 div op2;
  5: result:=op1 mod op2;
 end;
except
 on EDivByZero do
  if ListBoxOperation.ItemIndex=4 then
   result:=signe(op1)*Maxint
  else result:=op1;
 on EZeroDivide do
  result:=signe(op1)*Maxint;
 on EIntOverFlow do
  result:=signe(op1)*signe(op2)*Maxint;
end;

Editresult.text:=inttostr(result);
Editmessage.text:=inttostr(op1)+Toper[ListBoxOperation.ItemIndex]
      +' '+inttostr(op2)+'= '+Editresult.text;
end;
```

Le lecteur modifiera les choix de valeurs par défaut dans les gestionnaires d'exception. Dans l'exemple plus haut, ces choix ne sont qu'indicatifs et servent à montrer que l'on peut, soit arrêter un calcul, soit le continuer avec une valeur de remplacement après signalement de l'erreur. Il s'assurera par lui-même que la conjonction entre les plans d'action et les exceptions est un système de programmation défensive efficace.

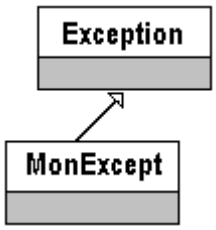
Créer et lancer ses propres exceptions

- Il est possible de construire de nouvelle classe d'exceptions personnalisées en héritant d'une des classes de Delphi mentionnées plus haut, ou à minima de la classe de base **Exception**.(cette classe contient une propriété public **property** Message: **string**, qui contient en type string le texte à afficher dans la boîte de dialogue des exceptions quand l'exception est déclenchée).

- Il est aussi possible de lancer une exception personnalisée (comme si c'était une exception propre à Delphi) à n'importe quel endroit dans le code d'une méthode d'une classe en instanciant un objet d'exception personnalisé et en le préfixant du mot clef **raise**.

- Le mécanisme général d'interception des exceptions à travers des gestionnaires d'exceptions **try**…**except** s'applique à tous les types d'exceptions y compris les exceptions personnalisées.

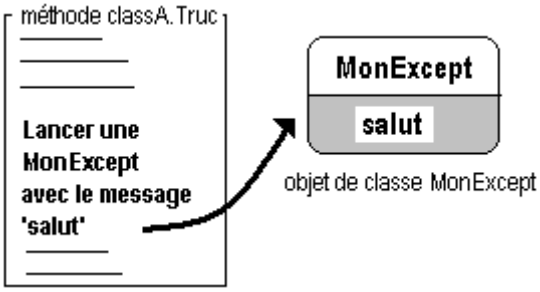
Création d'une nouvelle classe




```
Type
  MonExcept = class (Exception)

  End;
```

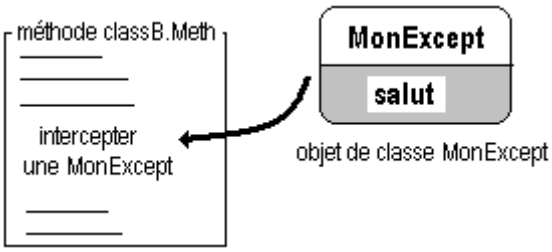
MonExcept hérite par construction de la propriété public **Message** de sa mère.

Lancer une MonExcept




```
Type
  MonExcept = class (Exception)

  End;

Procedure classA.Truc;
begin
   …..
 raise MonExcept.Create ( 'salut' );
  …..

end;
```

intercepter une MonExcept




```
MonExcept = class (Exception)…    end;

Procedure classB.Meth;
begin
  …..
   try….. // code à protéger
   except on MonExcept do begin
              ….. // code de réaction à l'exception
            end;
    end;
end;
```

Exercices chapitre 5

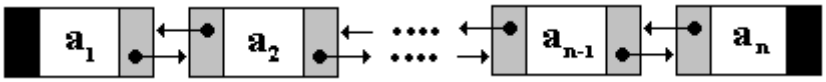
Ex-1 : Il est demandé de construire une classe de pile lifo ClassLifo héritant d'une Tlist (Un objet Tlist de Delphi, stocke un tableau de pointeurs, utilisé ici pour gérer une liste d'objets) et qui est réactive à l'empilement et au dépilement d'un objet. Nous proposons de suivre la démarche de la notice méthodologique du cours en nous inspirant de son code final pour construire deux événements dans la pile lifo et lui permettre de réagir à ces deux événements.

Ex-2 : Nous souhaitons développer rapidement un petit éditeur de texte qui nous permettra :

- de **lire** du texte à partir d'un fichier sur disque ou sur disquette (*ouvrir*),
- de **taper** du texte (*nouveau*),
- de **visualiser** le texte entré,
- d'effectuer sur ce texte les opérations classiques de *copier/ coller/ couper*,
- de le **sauvegarder** sur disque ou sur disquette (*enregistrer*).

Ex-3 : Nous reprenons la classe déjà construite ClassLifo de pile lifo héritant d'une Tlist réactive à l'empilement et au dépilement d'un objet, il est demandé de la rendre plus robuste en lui permettant lorsque la pile est vide de lancer une exception dépilement impossible.

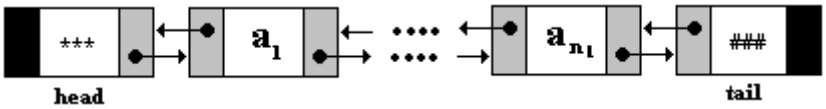
Ex-4 : On définit la structure de données de liste chaînée double, qui est une liste chaînée pouvant se parcourir dans les deux sens, chaque maillon de la liste est lié à son suivant et à son précédent sauf les maillons situés aux extrémités de la liste; (a1, … , an) sont les données de la liste :



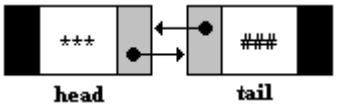
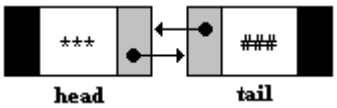
Implanter en delphi la classe TListeDble représentant une telle liste chaînée double et TcellDble un maillon de la liste (un maillon est donc un objet de calesse TcellDble), l'information du maillon est une chaîne de caractères.

TCellDble = **class**
  public
  info:**string**;
  constructor Créer (avant , apres:TCellDble; elt:string);
  procedure InsererAvant (cell:TCellDble);
  procedure InsererApres (cell:TCellDble);
  private
  next:TCellDble;
  prec:TCellDble;
  end;
 	
Les méthodes InsererAvant et InsererApres réalisent l'insertion (avant l'objet ou après l'objet)
	TlisteDble =**class**
	  public
	  constructor Create;
	  destructor Liberer;
	  procedure AjouterLeft (elt:string);
	  procedure AjouterRight (elt:string);
	  procedure InsererLeft (rang:integer;elt:string);
	  procedure InsererRight (rang:integer;elt:string);
	  procedure SupprimerLeft (rang:integer);
	  procedure SupprimerRight (rang:integer);
	  function ElementFromLeft (rang:integer):string;
	  function ElementFromRight (rang:integer):string;
	  function IndexFromLeftOf (elt:string):integer;
	  function IndexFromRightOf (elt:string):integer;
	  function Tete:TCellDble;
	  function Fin:TCellDble;
	  function Longueur : integer;
	  procedure Clear;
	  function ParcourirLeft:string;
	  private
	  head , tail : TCellDble;
	  end;

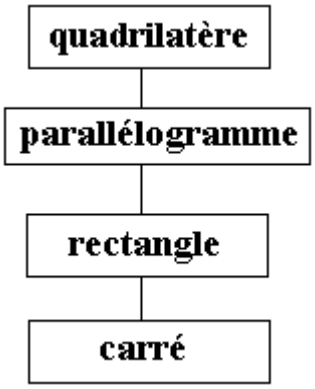
On propose pour simplifier l'écriture des algorithmes de parcours de la liste de mettre deux sentinelles ***head*** et ***tail*** bornant les deux "bouts" de la liste (cellules ne contenant de données significatives) :



Spécifications des méthodes à implanter :

constructor Create;	Crée une liste vide : 
destructor Liberer;	Libère la mémoire utilisée par la liste.
procedure AjouterLeft (elt : **string**);	Ajoute la donnée elt: **string** après head.
procedure AjouterRight (elt : **string**);	Ajoute la donnée elt: **string** avant tail.
procedure InsererLeft (rang:integer ; elt:**string**);	Insère la donnée elt: **string** au rang : integer , rang compté à partir de head (parcours à gauche).
procedure InsererRight (rang:integer ; elt:**string**);	Insère la donnée elt: **string** au rang : integer , rang compté à partir de tail (parcours à droite).
procedure SupprimerLeft (rang:integer);	Supprime la donnée de rang : integer , comptée à partir de head (parcours à gauche).
procedure SupprimerRight (rang:integer);	Supprime la donnée de rang : integer , comptée à partir de tail (parcours à droite).
function ElementFromLeft (rang:integer):**string**;	Renvoie la donnée de rang : integer , comptée à partir de head (parcours à gauche).
function ElementFromRight (rang:integer):**string**;	Renvoie la donnée de rang : integer , comptée à partir de tail (parcours à droite).
function IndexFromLeftOf (elt : **string**):integer;	Renvoie le rang de la position de la donnée elt : **string** compté à partir de head (parcours à gauche).
function IndexFromRightOf (elt : **string**):integer;	Renvoie le rang de la position de la donnée elt: **string** compté à partir de tail (parcours à droite).
function Tete:TCellDble;	Renvoie une référence sur head.
function Fin:TCellDble;	Renvoie une référence sur tail.
function Longueur : integer;	Fournit le nombre d'éléments utiles de la liste.
procedure Clear;	Remet la liste à vide : 
function ParcourirLeft : **string**;	Parcours des chaînes de la liste de head à tail en les concaténant entre elles et renvoie la chaîne unique obtenue.

Ex-5 : On définit la hiérarchie suivante dans les figures géometriques :



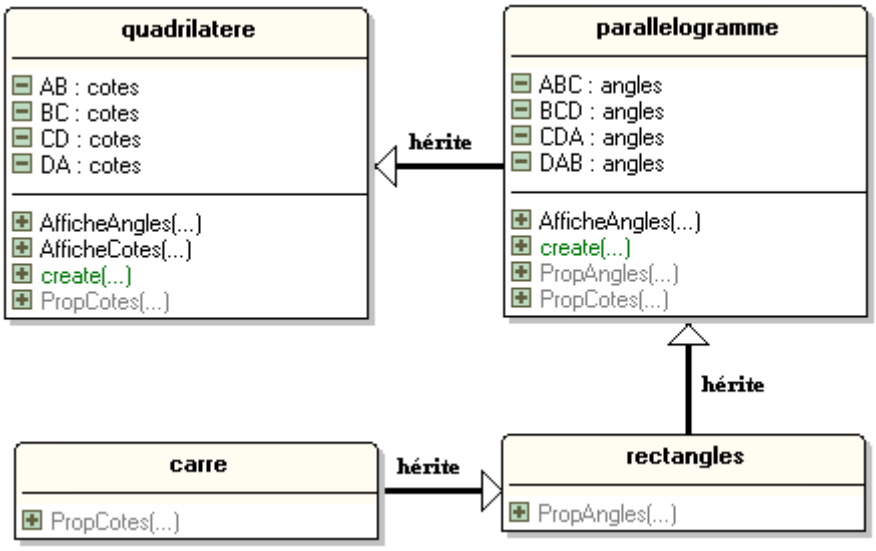
Un quadrilatère est une figure (A,B,C,D) possédant quatre côtés.

Un parallélogramme est un quadrilatère dont les côtés opposés sont parallèles et les angles opposés égaux.

Un rectangle est un parallélogramme dont tous les angles ont la même mesure : 90°

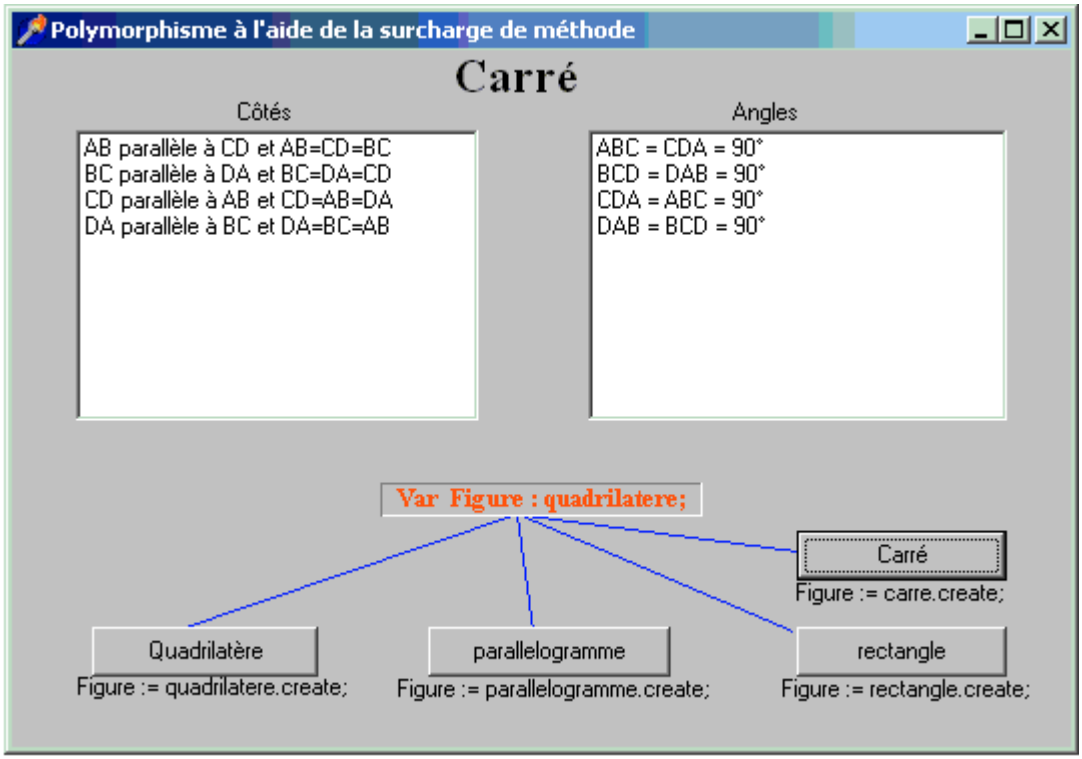
Un carré est un rectangle dont tous les côtés sont égaux.

Il est demandé d'implanter en Delphi la hiérarchie de classe selon les diagrammes UML ci-après :



Spécifications des méthodes à implanter	
type cotes = string; angles = string;	Les noms des côtés ou des sous forme de **string** angles (côtés :"AB", "CD", … ou angles : "ABC", "BCD",…).
function PropAngles (x:angles) **:** string;	Renvoie dans une **string** les propriétés d'un angle x : angles.
function PropCotes (x:cotes) **:** string;	Renvoie dans une **string** les propriétés d'un côté x : cotes.
procedure AfficheCotes (List **:** TlistBox);	Affiche dans un TlistBox les propriétés des 4 côtés.
procedure AfficheAngles (List **:** TlistBox);	Affiche dans un TlistBox les propriétés des 4 angles.

Il est demandé de construire une interface visuelle de test d'un objet de classe quadrilatère instancié à la demande selon l'une des trois classes filles (exemple ci-dessous d'une instanciation d'un quadrilatère en carré):



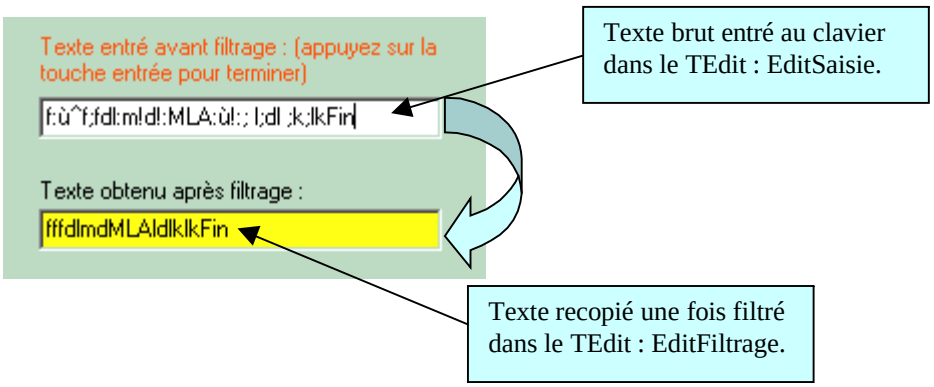
Ex-6 : Vous aurez à utiliser la classe TTimer qui encapsule les fonctions de timer de l'API Windows, on rappelle les propriétés utiles que cette classe contient :

```
property Enabled: Boolean; // Détermine si le timer répond aux événements timer.
property Interval: Cardinal; // Détermine l'intervalle de temps, exprimé en millisecondes, s'écoulant avant que le
composant timer génère un autre événement OnTimer.
property OnTimer: TNotifyEvent; // Se produit quand le temps spécifié par la propriété Interval s'est écoulé.

unit UClassclickMemo;
// classe TMemoFlash

interface
uses stdctrls, extctrls, Graphics, classes, controls;

 type
 TMemoFlash = class (…)
     public
      …
     private
      …
  end;

implementation

end.
```

Questions :

Construire une classe visuelle complète TMemoFlash héritée des TMemo qui clignote 10 fois (il changera de couleur jaune-bleu, par intermittence de 100ms entre chaque flash) lorsque l'utilisateur clique avec la souris dans le composant. Un TMemoFlash doit avoir automatiquement comme parent son propriétaire et lors de sa création il affichera l'adresse mémoire de la fenêtre de son parent et celle de sa propre fenêtre.

Ex-7 : construire une IHM de filtrage d'un texte entré au clavier dans un TEdit et recopié dans un TMemo une fois filtré :



Le filtrage consiste à ne conserver dans EditFiltrage que les lettres majuscules et minuscules, et à exclure tout autre caractère lors de la recopie du texte entré dans EditSaisie.

Le filtrage doit s'effectuer à la volée (c'est à dire au fur et à mesure que l'on tape du texte au clavier dans EditSaisie) et lorsque l'utilisateur appui sur la touche entrée du clavier le texte qui a été filtré doit être rangé dans un TMemo nommé MemoApresFiltrage. Prévoir deux versions possibles selon que l'utilisateur est autorisé à utiliser la touche d'effacement arrière (backspace) ou non dans la saisie du texte et utiliser l'événement OnKeypress pour effectuer le filtrage à la volée.

Ex-1 Code solution pratique : une pile Lifo événementielle

```
unit ULifoEvent ;

 interface
 uses classes,Dialogs ;

 type
  DelegateLifo = procedure ( Sender: TObject ; s :string ) of object ;

  ClassLifo = class (TList)
  private
     FOnEmpiler : DelegateLifo ;
     FOnDepiler : DelegateLifo ;
  public
    function Est_Vide : boolean ;
    procedure Empiler (elt : string ) ;
    procedure Depiler ( var elt : string ) ;
    property OnEmpiler : DelegateLifo read FOnEmpiler write FOnEmpiler ;
    property OnDepiler : DelegateLifo read FOnDepiler write FOnDepiler ;
end;

ClassUseLifo = class
 public
  procedure EmpilerListener( Sender: TObject ; s :string ) ;
  procedure DepilerListener( Sender: TObject ; s :string ) ;
  constructor Create ;
  procedure main ;
end;

implementation

 procedure ClassLifo.Depiler( var elt : string ) ;
 begin
  if not Est_Vide then
  begin
    elt :=string (self.First) ;
    self.Delete(0) ;
    self.Pack ;
    self.Capacity := self.Count ;
    if assigned(FOnDepiler) then
      FOnDepiler ( self ,elt )
  end
 end;

 procedure ClassLifo.Empiler(elt : string ) ;
 begin
  self.Insert(0 , PChar(elt)) ;
  if assigned(FOnEmpiler) then
     FOnEmpiler ( self ,elt )
 end;
```

Le type de l'événement : type pointeur de méthode (2 paramètres)

Champs privés stockant la valeur de l'événement (pointeur de méthode).

Evénement OnEmpiler :
- pointeur de méthode.

Evénement OnDepiler :
- pointeur de méthode.

self = la pile Lifo

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnDepiler **pointe vers elle, il est donc non nul.**
Sinon FOnDepiler **est nul (non assigné)**

L'instruction FOnDepiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnDepiler **pointe.**

Si une méthode dont la signature est celle du type DelegateLifo **est liée (gestionnaire de l'événement OnDepiler), le champ** FOnEmpiler **pointe vers elle, il est donc non nul.**
Sinon FOnEmpiler **est nul (non assigné)**

L'instruction FOnEmpiler (self ,elt) **sert à appeler la méthode vers laquelle** FOnEmpiler **pointe.**

```
function  ClassLifo.Est_Vide : boolean ;
begin
  result :=  self.Count = 0 ;
end;

 { ClassUseLifo }

constructor  ClassUseLifo.Create ;
begin
  inherited;
end;

procedure  ClassUseLifo.DepilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a depile : ' ,s) ;
end;

procedure  ClassUseLifo.EmpilerListener( Sender:  TObject ;  s :string ) ;
begin
  writeln ( 'On a empile : ' ,s) ;
end;

procedure  ClassUseLifo.main ;
var
  pileLifo : ClassLifo ;
  ch :string ;
 begin
   pileLifo := ClassLifo.Create ;
   pileLifo.OnEmpiler :=  EmpilerListener ;
   pileLifo.OnDepiler :=  DepilerListener ;
   pileLifo.Empiler( '[ eau ]' ) ;
   pileLifo.Empiler( '[ terre ]' ) ;
   pileLifo.Empiler( '[ mer ]' ) ;
   pileLifo.Empiler( '[ voiture ]' ) ;
   writeln ( 'Depilement de la pile :' ) ;
   while  not  pileLifo.Est_Vide  do
   begin
     pileLifo.Depiler(ch) ;
     writeln (ch) ;
   end;
   writeln ( 'Fin du depilement.' ) ;
   readln ;
 end;

end.
```

Classe de test créant une pile Lifo

Gestionnaire de l'événement OnDepiler : signature compatible avec DelegateLifo**.**

Gestionnaire de l'événement OnEmpiler : signature compatible avec DelegateLifo**.**

Méthode main **de test**

Instanciation d'une pile Lifo à tester.

Affectation de chaque gestionnaire à l'événement qu'il est chargé de gérer.

Empilement de 4 éléments

Dépilement de toute la pile

Application console **Project2.dpr** instanciant un objet de ClassUseLifo et lançant le test de la pile lifo par invocation de la méthode main.

```
program Project2;

{$APPTYPE CONSOLE}

uses   SysUtils , UlifoEvent ;
var execLifo : ClassUseLifo;
begin
  execLifo := ClassUseLifo.Create;
  execLifo.main
end.
```

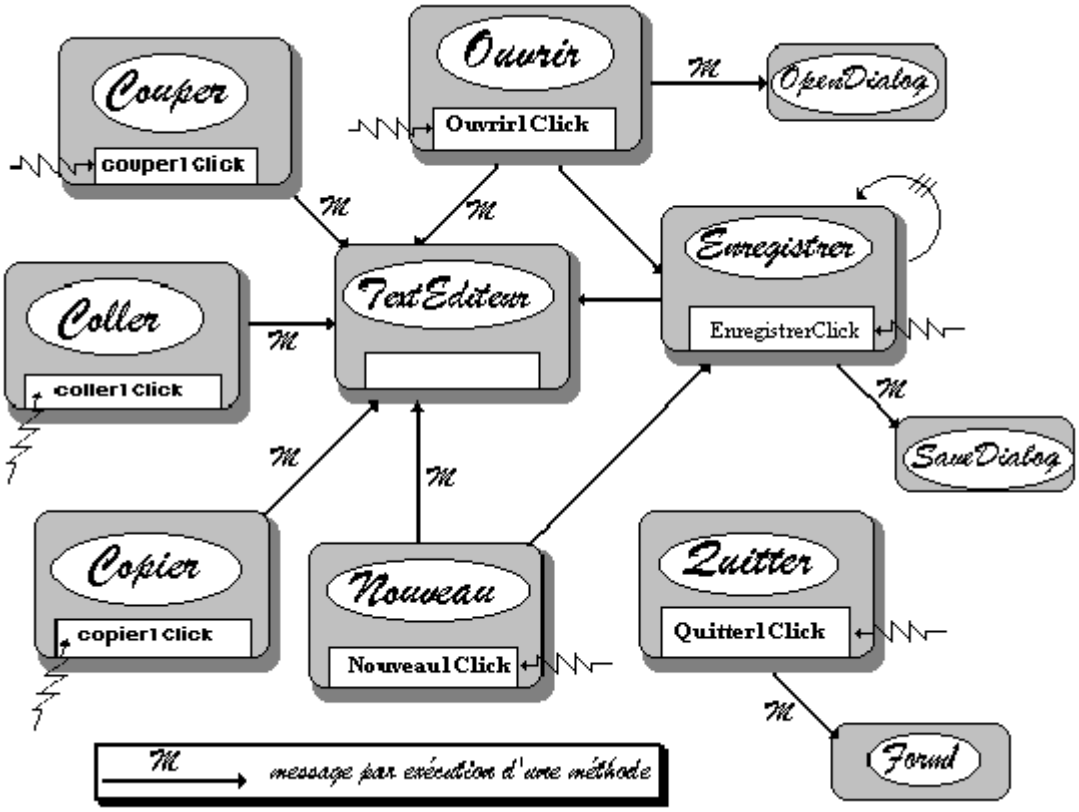
exécution

```
C:\Program Files\Borland\Delphi7\Bin\Project
On a empile : [ eau ]
On a empile : [ terre ]
On a empile : [ mer ]
On a empile : [ voiture ]
Depilement de la pile :
On a depile : [ voiture ]
[ voiture ]
On a depile : [ mer ]
[ mer ]
On a depile : [ terre ]
[ terre ]
On a depile : [ eau ]
[ eau ]
Fin du depilement.
```

Ex-2 Solution détaillée : un éditeur de texte

1. Les choix à partir de l'énoncé

Les objets potentiels réagiront à ces événements selon le graphe ci-dessous



L'objet ***TextEditeur*** est l'objet central sur lequel agissent tous les autres objets, il contiendra le texte à éditer. En faisant ressortir les opérations à effectuer, l'utilisation de la notion d'abstraction est naturelle.

Nous envisageons les actions suivantes obtenues par réaction à un événement Click de souris (choix des objets du graphe précédent):

nouveau *(utilise un objet à définir)*
couper *(utilise un objet à définir)*
copier *(utilise un objet à définir)*
coller *(utilise un objet à définir)*
ouvrir *(utilise un objet de dialogue)*
enregistrer *(utilise un objet de dialogue)*
quitter *(utilise un objet prédéfini Form)*

2. Les objets de composants

Delphi étant orienté objet nous allons exploiter complètement l'analyse présentée dans le graphe événementiel ci-haut en mettant en place quatre objets du genre composants visuels ou non visuels de Delphi.

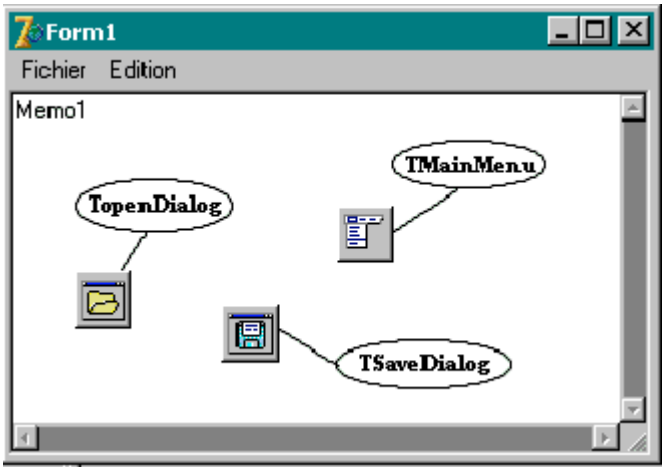
L'interface choisie

*Une fiche (**Form1**) comportant :*

- Un Tmemo avec deux ascenseurs

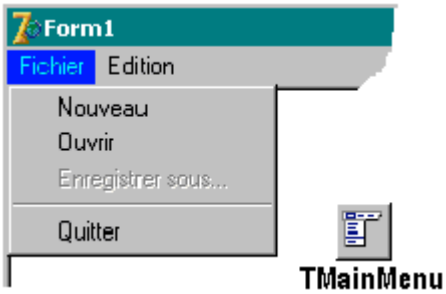
Trois composants non visuels :

- **TMainmenu** (une barre de 2 menus)
- **TopenDialog** (pour le chargement de fichier)
- **TSaveDialog** (pour la sauvegarde d'un texte)



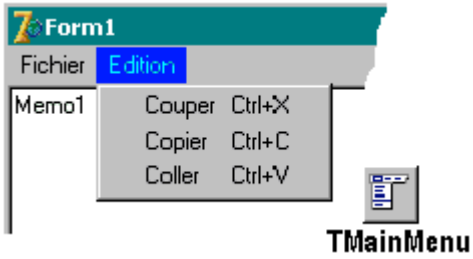
Le composant TMainmenu(1er menu)

comporte un premier menu *Fichier à 5 items*

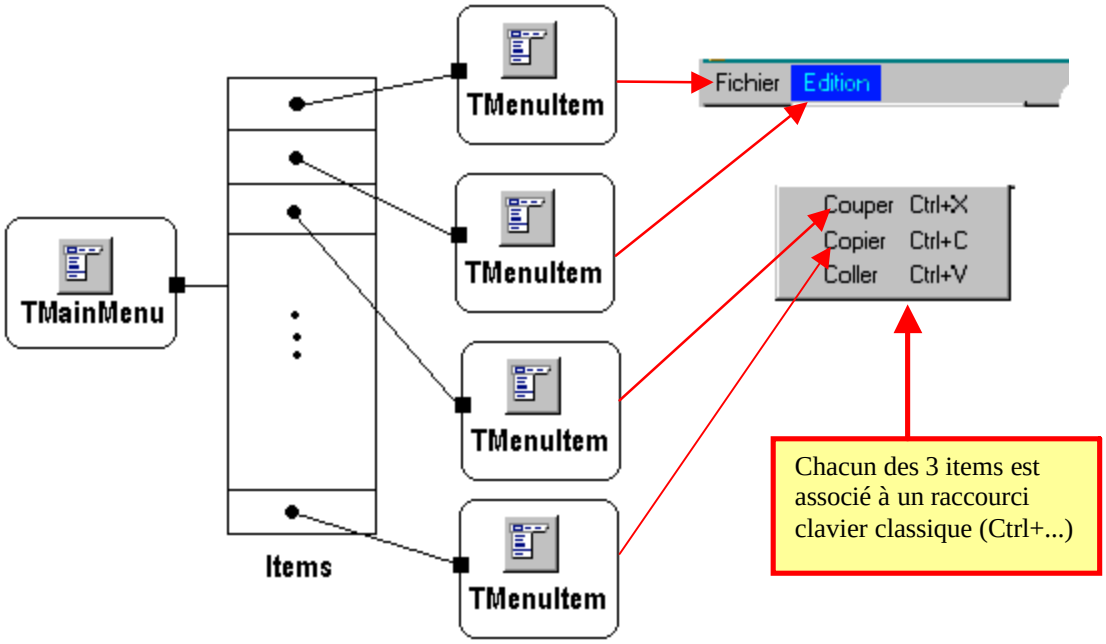


Le composant TMainmenu (2ème menu)

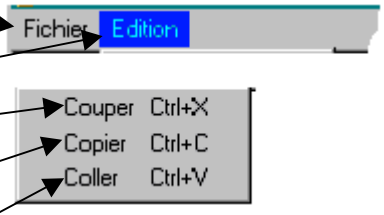
comporte un deuxième menu *Edition à 3 items.*



Les menus dans la barre et les sous-menus sont tous des objets de classe TmenuItem qui sont gérés à travers le champ Items d'un objet de classe TMainMenu :




```
object MainMenu1: TMainMenu
  ….
  object fichier1: TMenuItem
    Caption = 'fichier'
  end
  object edition1: TMenuItem
    Caption = 'edition'
    object couper1: TMenuItem
      Caption = 'couper'
    end
    object copier1: TMenuItem
      Caption = 'copier'
    end
    object coller1: TMenuItem
      Caption = 'coller'
    end
  end
end …..
```



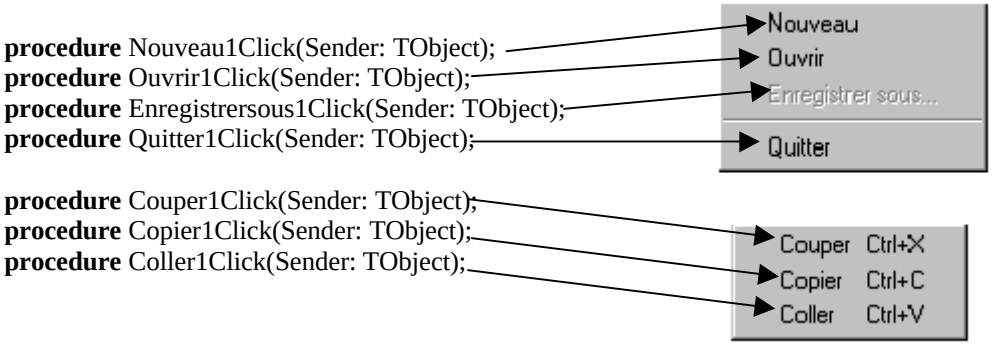
Correspondance entre le code caché par Delphi et l'inclusion des objets de TmenuItem dans un TmainMenu.

Mise en place de la gestion des événements.

A chacun des items utilisables de chacun des 2 menus, il nous faut associer un gestionnaire d'événement qui indique au programme comment il doit réagir lorsque l'utilisateur sélectionne un champ de l'un des menus par un click de souris. Nous avons vu que Delphi contient le mécanisme d'association des événements à nos gestionnaires. Beaucoup de contrôles (classes visuelles)et beaucoup d'autres classes non visuelles comme les TMenuItem, de Delphi sont sensibles à l'événement. de click de souris : **property** OnClick : TnotifyEvent.

Les gestionnaires de l'événement OnClick pour les TMenuItem

Voici les en-têtes des 7 gestionnaires de OnClick automatiquement construits :



Voici pour chaque gestionnaire le code écrit par le programmeur.

Le code du gestionnaire *Quitter1Click*

```
procedure TForm1.Quitter1Click(Sender: TObject);
begin
  Close; //écrit par le développeur (fermeture de la fenêtre)
end;
```

Le code du gestionnaire *Ouvrir1Click*

```
procedure TForm1.Ouvrir1Click(Sender: TObject);
begin
  if  OpenDialog1.Execute then //écrit par le développeur
  begin
    Enregistrersous1.enabled:=true;
    TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
  end
end;
```

Le code du gestionnaire *Enregistrersous1Click*

```
procedure TForm1.Enregistrersous1Click(Sender: TObject);
begin
  if  SaveDialog1.Execute then // écrit par le développeur
  begin
    Enregistrersous1.enabled:=false;
    TextEditeur.Lines.SaveToFile( SaveDialog1.FileName );
  end
end;
```

Le code du gestionnaire *Couper1Click*

```
procedure TForm1.Couper1Click(Sender: TObject);
begin
  TextEditeur.CutToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Copier1Click*

```
procedure TForm1.Copier1Click(Sender: TObject);
begin
  TextEditeur.CopyToClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Coller1Click*

```
procedure TForm1.Coller1Click(Sender: TObject);
begin
  TextEditeur.PasteFromClipboard; //écrit par le développeur
end;
```

Le code du gestionnaire *Nouveau1Click*

```
procedure TForm1.Nouveau1Click(Sender: TObject);
begin
  TextEditeur.Clear; &not; écrit par le développeur
  Enregistrersous1.enabled:=true &not; écrit par le développeur
end;
```

Il est à noter que nous avons écrit au total 16 lignes de programme Delphi pour construire notre micro-éditeur. Ceci est rendu possible par la réutilisabilité de méthodes déjà intégrées dans les objets de Delphi et en fait présentes dans le système Windows.

Vous pouvez voir la puissance de ce **RAD** lorsque vous observez par exemple l'instruction qui a permis de faire exécuter le "copier" ou le "chargement d'un fichier" :

```
TextEditeur.CopyToClipboard;
```

La méthode **CopyToClipboard** s'applique à l'objet **TextEditeur** qui est de la classe TMemo; cette instruction correspond à un ensemble complexe d'opérations de bas niveau :
le TMemo autorise une suite complexe d'opérations permettant de sélectionner un texte écrit sur plusieurs lignes du TMemo. Il les affiche en surbrillance et la méthode **CopyToClipboard** récupère le texte sélectionné puis le recopie enfin dans le presse-papier du système.

```
TextEditeur.Lines.LoadFromFile(OpenDialog1.FileName);
```

Nous n'avons par ailleurs eu aucun code particulier à écrire pour le chargement de fichier texte, la méthode **LoadFromFile** présente dans l'objet **Lines** (de classe TStrings) effectue toutes les actions.

A titre de travail personnel il est recommandé d'enrichir ce micro-éditeur avec la possibilité de changer la police de caractère, la couleur du fond etc…

Création d'une nouvelle classe

```
PileVideException = class (Exception)
end;
```

Lancer une PileVideException

```
procedure ClassLifo.Depiler( var elt : string ) ;
 begin
  if not Est_Vide then
  begin
   ……
  end
  else raise PileVideException.Create ('Impossible de dépiler: la pile est vide' )
 end;
```

Code complet de la pile

La classe Exception est dans la unit : SysUtils

La classe d'exception personnalisée : PileVideException

```
unit ULifoEvent ;

 interface
 uses classes,Dialogs, SysUtils ;

 type
  DelegateLifo = procedure ( Sender: TObject ; s :string ) of object ;

  PileVideException = class (Exception)
  end;

 ClassLifo = class (TList)
 private
    FOnEmpiler : DelegateLifo ;
    FOnDepiler : DelegateLifo ;
 public
   function Est_Vide : boolean ;
   procedure Empiler (elt : string ) ;
   procedure Depiler ( var elt : string ) ;
   property OnEmpiler : DelegateLifo read FOnEmpiler write FOnEmpiler ;
   property OnDepiler : DelegateLifo read FOnDepiler write FOnDepiler ;
end;

ClassUseLifo = class
 public
  procedure EmpilerListener( Sender: TObject ; s :string ) ;
  procedure DepilerListener( Sender: TObject ; s :string ) ;
  constructor Create ;
  procedure main ;
end;

implementation
```

```
procedure  ClassLifo.Depiler( var  elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
    elt :=string (self.First) ;
    self.Delete(0) ;
    self.Pack ;
    self.Capacity := self.Count ;
    if  assigned(FOnDepiler)  then
      FOnDepiler ( self ,elt )
   end
   else raise PileVideException.Create ('Impossible de dépiler: la pile est vide' )
  end;

 procedure  ClassLifo.Empiler(elt : string ) ;
 begin
   self.Insert(0 , PChar(elt)) ;
   if  assigned(FOnEmpiler)  then
       FOnEmpiler ( self ,elt )
 end;

 function  ClassLifo.Est_Vide : boolean ;
 begin
    result :=  self.Count = 0 ;
 end;

  { ClassUseLifo }

 constructor  ClassUseLifo.Create ;
 begin
    inherited;
 end;

 procedure  ClassUseLifo.DepilerListener( Sender:  TObject ;  s :string ) ;
 begin
   writeln ( 'On a depile : ' ,s) ;
 end;

 procedure  ClassUseLifo.EmpilerListener( Sender:  TObject ;  s :string ) ;
 begin
   writeln ( 'On a empile : ' ,s) ;
 end;

 procedure  ClassUseLifo.main ;
 var
   pileLifo : ClassLifo ;
   ch :string ;
  begin
    pileLifo := ClassLifo.Create ;
    pileLifo.OnEmpiler :=  EmpilerListener ;
    pileLifo.OnDepiler :=  DepilerListener ;
    pileLifo.Empiler( '[ eau ]' ) ;
    pileLifo.Empiler( '[ terre ]' ) ;
    pileLifo.Empiler( '[ mer ]' ) ;
    pileLifo.Empiler( '[ voiture ]' ) ;
    writeln ( 'Depilement de la pile :' ) ;
    while  not  pileLifo.Est_Vide  do
     begin
        pileLifo.Depiler(ch) ;
```

Instanciation d'un objet de type :
PileVideException
Et lancement de cet objet d'exception.

```
      writeln (ch) ;
    end;
  writeln ( 'Fin du depilement.' ) ;
  try
      pileLifo.Depiler(ch) ;
      writeln (ch) ;
   except on Ex : PileVideException do begin
            writeln ( Ex.Message ) ;
          end;
  end;
  readln ;
 end;

end.
```

On tente de dépiler alors que la pile a été entièrement vidée !

Interception d'un objet Ex de type : PileVideException
Et gestion de son Message.

Application console **Project2.dpr** instanciant un objet de ClassUseLifo et lançant le test de la pile lifo par invocation de la méthode main.

```
program Project2;

{$APPTYPE CONSOLE}

uses   SysUtils , UlifoEvent ;
var execLifo : ClassUseLifo;
begin
  execLifo := ClassUseLifo.Create;
  execLifo.main
end.
```

exécution

```
C:\Program Files\Borland\Delphi7\Bin\Project2.exe
On a empile : [ eau ]
On a empile : [ terre ]
On a empile : [ mer ]
On a empile : [ voiture ]
Depilement de la pile :
On a depile : [ voiture ]
[ voiture ]
On a depile : [ mer ]
[ mer ]
On a depile : [ terre ]
[ terre ]
On a depile : [ eau ]
[ eau ]
Fin du depilement.
Impossible de dúpiler: la pile est vide
```

Ex-4 Code solution pratique : liste chaînée double

```
unit UDblListChn;

 interface
```

```
type

 TCellDble = class
public
 info:string;
 constructor Creer(avant,apres:TCellDble;elt:string);
 procedure InsererAvant(cell:TCellDble);
 procedure InsererApres(cell:TCellDble);
private
 next:TCellDble;
 prec:TCellDble;
end;
```

```
TListeDble=class
 public
  constructor Create;
  destructor Liberer;
  procedure AjouterLeft(elt:string);
  procedure AjouterRight(elt:string);
  procedure InsererLeft(rang:integer;elt:string);
  procedure InsererRight(rang:integer;elt:string);
  procedure SupprimerLeft(rang:integer);
  procedure SupprimerRight(rang:integer);
  function ElementFromLeft(rang:integer):string;
  function ElementFromRight(rang:integer):string;
  function IndexFromLeftOf(elt:string):integer;
  function IndexFromRightOf(elt:string):integer;
  function Tete:TCellDble;
  function Fin:TCellDble;
  function Longueur:integer;
  procedure Clear;
  function ParcourirLeft:string;
 private
  head,tail:TCellDble;
  Long:integer;
  function CellFromLeftAt(rang:integer):TCellDble;
  function CellFromRightAt(rang:integer):TCellDble;

end;
```

```
implementation

 { TCellDble }
```

```
constructor TCellDble.Créer (avant,apres:TCellDble;
                             elt:string );
 begin
  self.next:=apres;
  self.prec:=avant;
  self.info:=elt;
 end;

 procedure TCellDble.InsererApres(cell:TCellDble);
 var cellNext:TCellDble;
 begin
  cellNext:= cell.next;
  self.next:=cellNext;
  cell.next:=self;
  cellNext.prec:=self;
  self.prec:=cell
 end;
```

```
procedure TCellDble.InsererAvant(cell:TCellDble);
 var cellPrec:TCellDble;
 begin
  cellPrec:= cell.prec;
  self.prec:=cellPrec;
  cell.prec:=self;
  cellPrec.next:=self;
  self.next:=cell
 end;
```

```
constructor TListeDble.Create; { TListeDble }
 begin
  head:=TCellDble.Créer (nil,nil,'***');
  tail:=TCellDble.Créer (nil,nil,'###');
  head.next:=tail;
  tail.prec:=head;
  Long:=0;
 end;

 destructor TListeDble.Liberer;
 begin
  self.Clear;
  self.head.Free;
  self.tail.Free;
 end;

 function TListeDble.ElementFromLeft(rang: integer):
string;
 var L,Loc:TCellDble;
  compte:integer;
  begin
   if (rang<=Long)and(rang>0) then
    result:=CellFromLeftAt(rang).info
   else
    result:='?'
  end;

 function TListeDble.ElementFromRight(rang:
integer): string;
 var L,Loc:TCellDble;
  compte:integer;
  begin
   if (rang<=Long)and(rang>0) then
    result:=CellFromRightAt(rang).info
   else
    result:='?'
  end;

function TListeDble.IndexFromLeftOf(elt: string):
integer;
 var i,index:integer;
  L:TCellDble;
  begin
   index:=0;
   L:=self.head;
   for i:=1 to long+1 do   begin
    if L.info=elt then
     break
    else
    begin
     L:=L.next;
     index:=index+1
    end
   end;
   if index>long then
    index:=0;
   result:=index
  end;
```

```
function TListeDble.IndexFromRightOf(elt: string):
integer;
 var i,index:integer;
  L:TCellDble;
  begin
   index:=0;
   L:=self.tail;
   for i:=1 to long+1 do   begin
    if L.info=elt then
     break
    else
    begin
     L:=L.prec;
     index:=index+1
    end
   end;
   if index>long then
    index:=0;
   result:=index
  end;

 procedure TListeDble.InsererLeft(rang: integer;
            elt: string);
 var Loc:TCellDble;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   Loc:=TCellDble.Creer(nil,nil,elt);
   Loc.InsererAvant(CellFromLeftAt(rang));
   Long:=Long+1;
  end
 end;

 procedure TListeDble.InsererRight(rang: integer; elt:
string);
 var Loc:TCellDble;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   Loc:=TCellDble.Creer(nil,nil,elt);
   Loc.InsererApres(CellFromRightAt(rang));
   Long:=Long+1;
  end
 end;

 procedure TListeDble.SupprimerLeft(rang: integer);
 var Loc,Lprec,Lnext:TCellDble;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   Loc:=CellFromLeftAt(rang);
   Lnext:=Loc.next;
   Lprec:=Loc.prec;
   Lnext.prec:=Lprec;
   Lprec.next:=Lnext;
   Loc.Free;
   Long:=Long-1
  end
 end;
```

```
procedure TListeDble.SupprimerRight(rang: integer);
 var Loc,Lprec,Lnext:TCellDble;
 begin
  if (rang<=Long)and(rang>0) then
  begin
   rang:=long-rang+1;
   self.SupprimerLeft(rang)
  end
 end;
function TListeDble.Fin: TCellDble;
 begin
  result:=self.tail
 end;

 function TListeDble.Tete: TCellDble;
 begin
  result:=self.head;
 end;

 procedure TListeDble.Clear;
 var L,Loc:TCellDble;
 begin
  if Long>0 then
  begin
   L:=self.head;
   while Assigned(L) do   begin
    Loc:=L;
    L:=L.next;
    if (Loc<>head)and(Loc<>tail)then
   //on n'efface pas la tête et la fin
    begin
     Loc.Free ;
    end
   end;
   head.next:=tail;
   tail.prec:=head;
   Long:=0;
  end
 end;

 function TListeDble.Longueur: integer;
 begin
  result:=Long
 end;

function TListeDble.ParcourirLeft:string;
 var L:TCellDble;
  sortie:string;
  begin
   L:=self.head;
   sortie:='';
   while Assigned(L) do   begin
    sortie:=sortie+L.info;
    L:=L.next;
   end;
   result:=sortie
  end;
```

```
procedure TListeDble.AjouterLeft(elt: string);
 var L,Loc:TCellDble;
 begin
  L:=self.head;
  Loc:=TCellDble.Creer(L,L.next,elt);
  L.next.prec:=Loc;
  L.next:=Loc;
  Long:=Long+1;
 end;

 procedure TListeDble.AjouterRight(elt: string);
 var L,Loc:TCellDble;
 begin
  L:=self.tail;
  Loc:=TCellDble.Creer(L.prec,L,elt);
  L.prec.next:=Loc;
  L.prec:=Loc;
  Long:=Long+1;
 end;

 function TListeDble.CellFromLeftAt(rang: integer):
TCellDble;
 var L,Loc:TCellDble;
  compte:integer;
  begin
   if (rang<=Long)and(rang>0) then
   begin
    L:=self.head;
    for compte:=1 to rang+1 do begin
   //la tete compte pour une cellule
     Loc:=L;
     L:=L.next;
    end;
    result:=Loc
   end
   else
    result:=nil
  end;

 function TListeDble.CellFromRightAt(rang: integer):
TCellDble;
 var L,Loc:TCellDble;
  compte:integer;
  begin
   if (rang<=Long)and(rang>0) then
   begin
    L:=self.tail;
    for compte:=1 to rang+1 do begin
   //la fin compte pour une cellule
     Loc:=L;
     L:=L.prec;
    end;
    result:=Loc
   end
   else
    result:=nil
  end;
end. // fin unit UDblListChn
```

Ex-5 Code solution pratique : hiérarchie de quadrilatères

```
unit UClassQuadrilat;

 interface
 uses stdctrls;
```

```
type
  cotes=string;
  angles=string;

  quadrilatere=class
 private
  AB,BC,CD,DA:cotes;
 public
  function PropCotes (x:cotes):string; virtual;
  //--statique car elle sert à tous :
  procedure AfficheCotes (List:TlistBox);
  procedure AfficheAngles (List:TlistBox); virtual;
  constructor create; virtual;
end;
```

```
Parallelogramme = class (quadrilatere)
 private
  ABC,BCD,CDA,DAB:angles;
 public
  function PropCotes(x:cotes):string;override;
  function PropAngles(x:angles):string; virtual;
  procedure AfficheAngles(List:TlistBox);override;
  constructor create;override;
end;

rectangles = class (parallelogramme)
//-- rectangle est déjà une function existante en Delphi!
 public
  function PropAngles(x:angles):string;override;
end;

carre = class (rectangles)
 public
  function PropCotes(x:cotes):string;override;
end;
```

```
implementation
```

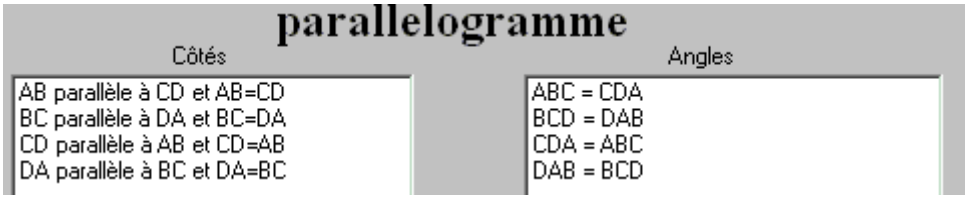
```
//////////////// CLASSE QUADRILATERE ////////////////
 constructor quadrilatere.create;
 begin
  AB:='AB';
  BC:='BC';
  CD:='CD';
  DA:='DA';
 end;

 function quadrilatere.PropCotes(x:cotes):string;
 begin
  result:=x+' : quelconque'
 end;

 procedure quadrilatere.AfficheCotes(List:TlistBox);
 begin
  List.Clear;
  List.Items.Add(PropCotes(AB));
  List.Items.Add(PropCotes(BC));
  List.Items.Add(PropCotes(CD));
  List.Items.Add(PropCotes(DA));
 end;

 procedure quadrilatere.AfficheAngles(List:TlistBox);
 begin
  List.Clear;
  List.Items.Add('angles quelconques')
 end;
```

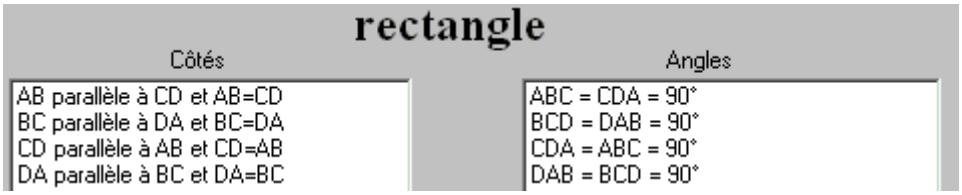
```
//////// CLASSE PARALLELOGRAMME  ///////////
 constructor parallelogramme.create;
 begin
  inherited;
  ABC:='ABC';
  BCD:='BCD';
  CDA:='CDA';
  DAB:='DAB';
 end;

 function parallelogramme.PropCotes(x:cotes):string;
 begin
  if x='AB' then
   result:=x+' parallèle à CD et AB=CD'
  else
   if x='BC' then
    result:=x+' parallèle à DA et BC=DA'
   else
    if x='CD' then
     result:=x+' parallèle à AB et CD=AB'
    else
     if x='DA' then
      result:=x+' parallèle à BC et DA=BC'
 end;
```

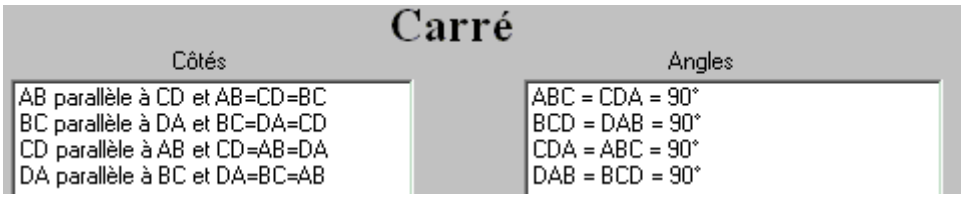
```
// CLASSE PARALLELOGRAMME  (suite) //
function parallelogramme.PropAngles
                                (x:angles):string;
 begin
  if x='ABC' then
   result:='ABC = CDA'
  else
   if x='BCD' then
    result:='BCD = DAB'
   else
    if x='CDA' then
     result:='CDA = ABC'
    else
     if x='DAB' then
      result:='DAB = BCD'
 end;

 procedure parallelogramme.AfficheAngles
                                (List:TlistBox);
 begin
  List.Clear;
  List.Items.Add(PropAngles(ABC));
  List.Items.Add(PropAngles(BCD));
  List.Items.Add(PropAngles(CDA));
  List.Items.Add(PropAngles(DAB));
 end;
```

```
/////////////// CLASSE RECTANGLE ////////////////
 function rectangles.PropAngles(x:angles):string;
 var s:string;
 begin
  s:=inherited PropAngles(x);
  result:=s+' = 90°'
 end;

/////////////// CLASSE CARRE ////////////////
 function carre.PropCotes(x:cotes):string;
 var s:string;
 begin
  s:=inherited PropCotes(x); {celui de parallélogramme:
première classe ancêtre où il est surchargé ou défini }
  if (x='AB') then
   result:=s+'=BC'
  else
   if(x='bc')then
    result:=s+'=CD'
   else
    if(x='cd')then
     result:=s+'=DA'
    else
     if(x='da') then
      result:=s+'=AB'
 end;

end.
```

Exemples d'affichages obtenus pour un objet quadrilatère instancié dans chaque type :







Ex-6 Code solution pratique : un TMemoFlash qui clignote

Une unité d'IHM qui instancie un TmemoFlash lorsque l'on clique sur le bouton Button1 :

```
unit UClassclickMemo;

 interface

 uses
  StdCtrls,ExtCtrls,
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

 type
  TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
 private
 { Private declarations }
 public
 { Public declarations }
end;

TMemoFlash=class(TMemo)
 private
  Tempo:TTimer;
  Timing:integer;
  procedure flashMemo(Sender : TObject);
  procedure ClickMemo(Sender : TObject);
 public
  constructor Create(AOwner: TComponent);
  destructor Destroy;
end;

 var
  Form1: TForm1;

implementation

 {$R *.dfm}
 constructor TMemoFlash.Create(AOwner: TComponent);
 begin
  inherited;
  self.Parent:=TWinControl(AOwner);
  self.Lines.Append('adresse Owner = '+inttostr((AOwner as TWincontrol).Handle));
  self.Lines.Append('adresse Memo = '+inttostr(self.Handle));
  Tempo:=TTimer.Create(AOwner);
```

```
 Tempo.Enabled:=false;
 Tempo.Interval:=100;
 Tempo.OnTimer:= flashMemo;
 self.OnClick:= ClickMemo;
end;

destructor TMemoFlash.Destroy;
begin
 Tempo.Free;
 inherited;
end;

procedure TMemoFlash.flashMemo(Sender : TObject);
begin
 Timing:=Timing+1;
 if Timing<10 then
  if self.color=clyellow then
   self.color:=claqua
  else
   self.color:=clyellow
 else
  tempo.enabled:=false
end;

procedure TMemoFlash.ClickMemo(Sender : TObject);
begin
 Timing:=0;
 Tempo.Enabled:=true;
end;

//-----------------------------------------------------------------------//
procedure TForm1.Button1Click(Sender: TObject);
var x:TMemoFlash;
begin
 x:=TMemoFlash.Create(self);
end;

end.
```

Ex-7 Code solution pratique : Filtrage avec KeyPress

```
unit UKeyPress;
//permet le filtrage de caractères en mode clavier à partir d'un TEdit
//(effacement possible seulement à la fin du texte entré)
interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

 type
  TForm1 = class(TForm)
  EditSaisie: TEdit;
  MemoApresFiltrage: TMemo;
  EditFiltre: TEdit;
  Label1: TLabel;
  Label2: TLabel;
```

```
 Label3: TLabel;
 procedure EditSaisieKeyPress(Sender: TObject; var Key: Char);
 procedure EditSaisieClick(Sender: TObject);
 procedure FormCreate(Sender: TObject);
private
{ Déclarations privées }
public
{ Déclarations publiques }
 LeTexte:string; //contient le texte entré au clavier
 procedure TexteSansBackSp(var Entree:string;car:char);
 procedure TexteAvecBackSp(var Entree:string;car:char);
end;

var
 Form1: TForm1;

implementation

{$R *.dfm}


procedure TForm1.TexteSansBackSp(var Entree:string;car:char);
//ne traite pas le backspace
begin
 if car<>#13 then
 begin
  if car in ['a'..'z']+['A'..'Z'] then
  begin
   Entree:=Entree+car;
   EditFiltre.Text:=Entree
  end
 end
 else
 begin
  MemoApresFiltrage.Lines.Add(Entree);
  Entree:='';
  EditSaisie.Text:=''
 end
end;
```

```
procedure TForm1.TexteAvecBackSp(var Entree:string;car:char);
//traitement du backspace à partir de la fin du texte
begin
 if (ord(car)=8)and(length(EditSaisie.Text)<>0) then
  if EditSaisie.Text[length(EditSaisie.Text)] in ['a'..'z']+['A'..'Z'] then
  begin
   delete(Entree,length(Entree),1); //on efface le dernier caractère
   EditFiltre.Text:=Entree        //on recopie le nouveau texte
  end;
 TexteSansBackSp(Entree,car)
end;

{-----------------------------------------------------------------------------------}
procedure TForm1.EditSaisieKeyPress(Sender: TObject; var Key: Char);
//filtrage des lettres non accentuées seulement
begin
 TexteSansBackSp(LeTexte,Key);
 //TexteAvecBackSp(LeTexte,Key);
end;
```

```
procedure TForm1.EditSaisieClick(Sender: TObject);
begin
 EditSaisie.SelStart:= length(EditSaisie.text) //replace systématiquement le curseur à la fin
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 EditSaisie.SetFocus
end;

End.
```

Chapitre 6 : Utiliser des grammaires pour programmer

6.1.Programmation avec les grammaires de type 2

- programmation par les grammaires
- C-grammaire et automate à pile de mémoire

6.2.Automates et grammaires de type 3

- automates pour les grammaires de type 3
- implantation d'un automate d'état fini en pascal
  - déterministe à partir des règles
  - déterministe à partir de sa table des transitions
  - non-déterministe à partir des règles
  - automate pour les identificateurs
  - automate pour les constantes numériques

6.3.Projet d'interpréteur de micro-langage

- la grammaire du micro-langage
- construction de l'analyseur
- construction de l'interpréteur

6.4.Projet d'indentateur de code Delphi

- spécifications
- architecture
- implantation

6.1 Programmation avec des C-grammaires

Plan du chapitre:

1. Programmation par les grammaires

1.1 Méthode pratique de programmation avec un langage récursif
1.2 Application de la méthode : un mini-français

2. C-grammaire et automate à pile de mémoire

2.1 Définition d'un automate à pile
2.2 Algorithme de fonctionnement d'un automate à pile
2.3 Programme Pascal d'un automate à pile

1. Programmation par les grammaires (programme en Delphi et Java)

D'un point de vue pratique, les grammaires sont un outil abstrait puissant. Nous avons vu qu'elles permettaient de décrire des langages de quatre catégories. Elles servent aussi :

- soit à générer des phrases dans le langage engendré par la grammaire (en ce sens elles permettent la programmation),
- soit à analyser un énoncé quelconque pour savoir s'il appartient ou non au langage engendré par la grammaire (en ce sens elles permettent la construction des analyseurs et des compilateurs).

Nous adoptons ici le point de vue " mode génération " d'une grammaire afin de s'en servir comme d'un outil de spécification sur les mots du langage engendré par cette grammaire. On appelle aussi cette démarche *programmation par la syntaxe*.

Nous nous restreindrons au C-grammaires et aux grammaires d'états finis.

Soit $G = (V_N,V_T,S,R)$, une telle grammaire et L(G) le langage engendré par G.

Objectif : Nous voulons construire un programme qui nous exhibe sur l'écran des mots du langage L(G).

1.1 Méthode pratique de programmation avec un langage récursif

$G = (V_N,V_T,S,R)$ → traduction en programme pratique en langage X générateur de mots.

La grammaire G est supposée ne pas contenir de règle récursive gauche (du genre $A \rightarrow A\alpha$), sinon il faut essayer de la changer ou abandonner.

1° Tous les éléments du vocabulaire auxiliaire V_N deviennent les noms d'un bloc-procédure du programme.

2° Le vocabulaire terminal V_T est décrit soit par un type prédéfini du langage X s'il est simple, sinon par une structure de donnée et un TAD.

3° Toutes les règles de G sont traduites de cette manière :

3.1° le symbole de V_N de la partie Gauche de la règle indique le nom du bloc-procédure que l'on va implanter.
3.2° la partie droite d'une règle correspond à l'implémentation du corps du bloc-procédure, pour chaque symbole α de cette partie droite si c'est :

- un élément de V_T, il est traduit par une écriture immédiate de sa valeur (généralement un **écrire**(α) traduit dans le langage X).
- un élément de V_N, il est traduit par un appel au bloc-procédure du même nom que lui.

4° Le bloc-procédure représentant l'axiome S est appelé dans le programme principal. Chaque appel de S fournira un mot du langage L(G).

Afin de bien persuader le lecteur de la non dépendance de la méthode vis à vis du langage nous construisons l'exemple en parallèle en Delphi et en Java.

Exemple fictif :

grammaire	*Traduction en Delphi*	*Traduction en Java*
G = (V_N,V_T,S, R) V_N = { S, A, B } V_T = { a, b } **Axiome** : S **Règles** : ..... k : S → aAbBb	V_N → **procedure** S ; V_T → **Type** Vt = char ; **procedure** A ; **procedure** B ;	V_N → **void** S( ) ; V_T → **char** ; **void** A( ) ; **void** B( ) ;

La règle k est traduite par l'implantation du corps du bloc-procédure associé à l'axiome S (partie gauche):

règle	*Traduction en Delphi*	*Traduction en Java*

k : S → aAbBb	**procedure** S ;
	begin
	writeln('a') ;
	A ;
	writeln('b') ;
	B ;
	writeln('b') ;
	end;
		void S ( ) {
		System.out.println('a');
		A( ) ;
		System.out.println('b');
		B( ) ;
		}

Le lecteur comprend ici le pourquoi de la contrainte de règles non récursives gauches (du genre A → A α), le bloc-procédure s'écrirait alors :

règle	*Traduction en Delphi*	*Traduction en Java*

A → A α	**procedure** A ;
	begin
	A ;
	...
	end ;
		void A ( )
		{
		A( ) ;
		...
		}

Ce qui conduirait le programme à un empilement récursif infini du bloc-procédure A (limité par la saturation de la pile d'exécution de la machine avec production d'une exception de débordement de pile).

1.2 Application de la méthode : un mini-français

Etant donné G une grammaire d'un sous-ensemble du français dénommé **mini-fr**.

G = (V_N,V_T,S,R)
V_T ={**le, un, chat, chien, aime, poursuit, malicieusement, joyeusement, gentil, noir, blanc, beau, ' . '**}
V_N = { ⟨*phrase*⟩, ⟨*GN*⟩, ⟨*GV*⟩, ⟨*Art*⟩, ⟨*Nom*⟩, ⟨*Adj*⟩, ⟨*Adv*⟩, ⟨*verbe*⟩ }
Axiome : ⟨ *phrase* ⟩

```
Règles :
1 : ⟨ phrase ⟩→⟨ GN ⟩⟨ GV ⟩⟨ GN ⟩ .
2 : ⟨ GN ⟩→⟨ Art ⟩⟨ Adj ⟩⟨ Nom ⟩
3 : ⟨ GN ⟩→⟨ Art ⟩⟨ Nom ⟩⟨ Adj ⟩
4 : ⟨ GV ⟩→⟨ verbe ⟩ | ⟨ verbe ⟩⟨ Adv ⟩
5 : ⟨ Art ⟩→ le | un
6 : ⟨ Nom ⟩→ chien | chat
7 : ⟨ verbe ⟩→ aime | poursuit
8 : ⟨ Adj ⟩→ blanc | noir | gentil | beau
9 : ⟨ Adv ⟩→ malicieusement | joyeusement
```

Traduisons à l'aide de la méthode précédente, cette grammaire G en un programme Delphi générant des phrases de L(G).

A) les procédures du programme

Chaque élément de $\mathbf{V_N}$ est associé à une procédure :

$\mathbf{V_N}$ = {⟨phrase⟩, ⟨GN⟩, ⟨GV⟩, ⟨Art⟩, ⟨Nom⟩, ⟨Adj⟩, ⟨Adv⟩, ⟨verbe⟩}

$\mathbf{V_N}$	*Traduction en Delphi*	*Traduction en Java*
$\mathbf{V_N}$ = {⟨phrase⟩, ⟨GN⟩, ⟨GV⟩, ⟨Art⟩, ⟨Nom⟩, ⟨Adj⟩, ⟨Adv⟩, ⟨verbe⟩}	**procedure** phrase; **procedure** GN; **procedure** GV; **procedure** Art; **procedure** Nom; **procedure** Adj; **procedure** Adv; **procedure** verbe;	**void** phrase() **void** GN() **void** GV() **void** Art() **void** Nom() **void** Adj() **void** Adv() **void** verbe()

B) les types de données associés à $\mathbf{V_T}$

Nous utilisons la structure de tableau de chaînes, commode à cause de sa capacité d'accès direct pour stocker les éléments de $\mathbf{V_T}$. Toutefois, au lieu de ne prendre qu'un seul tableau de chaînes pour $\mathbf{V_T}$ tout entier, nous partitionnons $\mathbf{V_T}$ en 5 sous-ensembles disjoints :

$\mathbf{V_T}$ = tnom ∪ tadjectif ∪ tarticle ∪ tverbe ∪ tadverbe
où :
tnom={ chat, chien } tadjectif={ blanc, noir, gentil, beau }
tarticle ={ le, un } tverbe ={ aime, poursuit }
tadverbe={ malicieusement, joyeusement }

Spécification d'implantation en Delphi :

Ces cinq ensembles sont donc représentés en Delphi chacun par un tableau de chaînes. Nous définissons le type **mot** comme le type général, puis cinq tableaux de type mot.

```
const
   Maxnbmot=4; // nombre maximal de mots dans un tableau
```

```
 type
    mot = array[1..Maxnbmot]of string;
 champs
    tnom, tadjectif, tarticle ,tverbe, tadverbe : mot;
```

Nous construisons une classe que nous nommons Gener_fr qui est chargée de construire et d'afficher une phrase du langage **mini-fr** :

Tous les champs seront privés la seule méthode publique est la méthode phrase qui traduit l'axiome de la grammaire et qui lance le processus de génération et bienentendu le constructeur d'objet de la classe qui est obligatoirement publique.

Etat de la classe à ce niveau de construction :

```
const
   Maxnbmot=4; // nombre maximal de mots dans un tableau
type
   mot = array[1..Maxnbmot]of string;

Gener_fr = class
  private
    tnom, tadjectif, tarticle, tverbe, tadverbe : mot;
    procedure GN;
    procedure GV;
    procedure Art;
    procedure Nom;
    procedure Adj;
    procedure Adv;
    procedure verbe;
 public
    constructor Creer; // constructeur d'objet
    procedure phrase; // axiome de la grammaire
 end;
```

Spécification d'implantation en Java : Les spécifications sont les mêmes qu'en Delphi

Etat de la classe Java à ce niveau de construction :

```
   class  Gener_fr {
     final int Maxnbmot=4; // nombre maximal de mots dans un tableau
     private   String[ ] tnom ;
     private   String[ ] tadjectif ;
     private   String[ ] tarticle ;
     private   String[ ] tverbe ;
     private   String[ ] tadverbe ;
     private void GN  ( ) {  }
     private void GV   ( ) {  }
     private void Art  ( ) {  }
     private void Nom  ( ) {  }
     private void Adj  ( ) {  }
     private void Adv  ( ) {  }
     private void verbe  ( ) {  }
     public void phrase   ( ) {  } // axiome de la grammaire
    }
```

C) Initialisation des données associées à V_T

Un ensemble de méthodes de chargement est élaboré afin d'initialiser les contenus des différents tableaux, ce qui permet de changer aisément leur contenu, voici dans la classe Gener_fr les méthodes Delphi associées :

procedure Gener_fr.initnom; **begin** tnom[1]:='chat'; tnom[2]:='chien'; **end**;	**procedure** Gener_fr.initverbe; **begin** tverbe[1]:='poursuit'; tverbe[2]:='aime'; **end**;
procedure Gener_fr.initadjectif; **begin** tadjectif[1]:='beau'; tadjectif[2]:='gentil'; tadjectif[3]:='noir'; tadjectif[4]:='blanc'; **end**;	**procedure** Gener_fr.initarticle; **begin** tarticle[1]:='le'; tarticle[2]:='un'; **end**;
procedure Gener_fr.initadverbe; **begin** tadverbe[1]:='malicieusement'; tadverbe[2]:='joyeusement'; **end**;	Delphi

Ces cinq méthodes sont appelées dans une méthode générale d'initialisation du vocabulaire $\mathbf{V_T}$ tout entier.

```
procedure Gener_fr.initabl;
begin
   initnom;
   initarticle;
   initverbe;
   initadjectif
end;
```

Initialisation des contenus en Java :

void initnom ( ) **{** tnom[0] ="chat"; tnom[1] = "chien"; **}**	**void** initverbe ( ) **{** tverbe[0] = "poursuit"; tverbe[1] = "aime"; **}**
void initadjectif ( ) **{** tadjectif[0] = "beau"; tadjectif[1] = "gentil"; tadjectif[2] = "noir"; tadjectif[3] = "blanc"; **}**	**void** initarticle ( ) **{** tarticle[0] = "le"; tarticle[1] = "un"; **}** Java
void initadverbe ( ) **{** tadverbe[0] = "malicieusement"; tadverbe[1] = "joyeusement"; **}**	**void** initabl ( ){ initnom ( ); initarticle ( ); initverbe ( ); initadjectif ( ); initadverbe ( ); **}**

Nous avons besoin d'une fonction de tirage aléatoire lorsqu'il se présente un choix à faire entre plusieurs règles dérivant du même élément de $\mathbf{V_N}$, comme dans la règle suivante :

règle 4 : 〈 ***GV*** 〉→〈 ***verbe*** 〉 **|** 〈 ***verbe*** 〉〈 ***Adv*** 〉

où nous trouvons deux cas de dérivation possible pour le groupe verbal **GV** :

soit < ***verbe*** >,
soit < ***verbe*** > < ***Adv*** >

Le programme devra procéder à un choix aléatoire entre l'une ou l'autre des dérivations possibles.

Nous construisons une méthode Alea qui reçoit en entrée un entier indiquant le nombre **n** de choix possibles et qui renvoie une valeur aléatoire comprise entre **1** et **n**.

Une implantation possible:

Delphi	Java

function Gener_fr.Alea(n:integer):integer;
begin
  result := trunc(random*100)mod n+1;
end;
	private Random ObjAlea = **new** Random();
	
	int Alea (**int** n) **{**
	  return ObjAlea.nextInt(n);
	}

D) Traduction de chacune des règles de G

Nous traduisons en employant la méthode proposée règle par règle.

REGLE

```
1 : 〈 phrase 〉→〈 GN 〉〈 GV 〉〈 GN 〉.
```

Nous construisons le corps de la méthode phrase qui est la partie gauche de la règle. Les instructions correspondent aux appels des procédures GN, GV.

Delphi	Java

procedure Gener_fr.phrase;
begin
  GN;
  GV;
  GN;
  writeln('.')
end;
	void phrase ( )
	{
	  GN ( );
	  GV ( );
	  GN ( );
	  System.out.println('.') ;
	}

REGLE

```
2 : 〈 GN 〉→〈 Art 〉〈 Adj 〉〈 Nom 〉
3 : 〈 GN 〉→〈 Art 〉〈 Nom 〉〈 Adj 〉
```

Nous traitons ensemble ces deux règles car elles correspondent à la même procédure de génération du groupe nominal **GN**.

Ici nous avons un tirage aléatoire à faire pour choisir laquelle des deux dérivations le programme utilisera.

Delphi	Java

procedure Gener_fr.GN;
begin
if Alea(2)=1 **then** *// pour règle 3*
begin
Art;
Nom;
Adj
end
else *// pour règle 2*
begin
Art;
Adj;
Nom
end
end ;
	void GN ( )
	{
	if (Alea(2) ==1) *// pour règle 3*
	{
	Art ( );
	Nom ( );
	Adj ( );
	}
	else *// pour règle 2*
	{
	Art ( );
	Adj ( );
	Nom ( );
	}
	}

REGLE

```
4 : 〈 GV 〉→〈 verbe 〉 | 〈 verbe 〉〈 Adv 〉
```

Dans ce cas nous avons aussi à faire procéder à un tirage aléatoire afin de choisir soit :

la dérivation 〈 GV 〉→〈 verbe 〉
ou bien la dérivation 〈 GV 〉→〈 verbe 〉〈 Adv 〉

Delphi	Java
procedure Gener_fr.GV; **begin** **if** Alea(2)=1 **then** *// règle: < verbe >* Verbe **else** *// règl:e < verbe >< Adv >.* **begin** Verbe; Adv **end** **end**;	**void** GV ( ) **{** **if** (Alea(2) ==1) *// règle: < verbe >* Verbe ( ); **else** *// règl:e < verbe >< Adv >.* **{** Verbe ( ); Adv ( ); **}** **}**

Les règles suivantes étant toutes des règles terminales, elles sont donc traitées comme le propose la méthode : chaque règle terminale est traduite par un writeln(a). Lorsqu'il y a plusieurs choix possibles, là aussi nous procédons à un tirage aléatoire afin d'emprunter l'une des dérivations potentielles.

REGLE

```
5 : 〈 Art 〉→ le | un
6 : 〈 Nom 〉→ chien | chat
7 : 〈 verbe 〉→ aime | poursuit
8 : 〈 Adj 〉→ blanc | noir | gentil | beau
9 : 〈 Adv 〉→ malicieusement | joyeusement
```

Delphi	Java
procedure Gener_fr.Art; **begin** write(tarticle[Alea(2)],' ') **end**;	**void** Art ( ) **{** System.out.print(tarticle[Alea(2)]+' ') ; **}**
procedure Gener_fr.Adj; **begin** write(tadjectif[Alea(4)],' ') **end**;	**void** Adj ( ) **{** System.out.print(tadjectif [Alea(4)]+' ') ; **}**
procedure Gener_fr.Verbe; **begin** write(tverbe[Alea(2)],' ') **end**;	**void** Verbe ( ) **{** System.out.print(tverbe [Alea(2)]+' ') ; **}**
procedure Gener_fr.Nom; **begin** write(tnom[Alea(2)],' ') **end**;	**void** Nom ( ) **{** System.out.print(tnom [Alea(2)]+' ') ; **}**
procedure Gener_fr.Adv; **begin** write(tadverbe[Alea(2)],' ') **end**;	**void** Adv ( ) **{** System.out.print(tadverbe [Alea(2)]+' ') ; **}**

Le programme principal se bornera à appeler la procédure **phrase** (l'axiome de la grammaire) à chaque fois que nous voulons engendrer une phrase. Ci-dessous dans le tableau de gauche nous listons la classe Gener_fr Delphi comportant toutes les méthodes précédentes et le programme d'instanciation d'un objet de cette classe permettant la génération aléatoire de phrases. A l'identique dans le tableau de droite, nous listons la classe Gener_fr Java, puis une autre classe principale générant une suite de phrases aléatoires :

Classe Delphi	Classe Java
unit UclassGenerFr; **interface** **const** Maxnbmot=4; **type** mot=array[1..Maxnbmot]**of** string; Gener_fr = **class** **private** tnom,tadjectif,tarticle,tverbe,tadverbe: mot; **procedure** initnom; **procedure** initverbe; **procedure** initadverbe; **procedure** initadjectif; **procedure** initarticle; **procedure** initabl; **function** Alea(n:integer):integer; **procedure** Article; **procedure** Nom; **procedure** Adjectif; **procedure** Adverbe; **procedure** Verbe; **procedure** fin; **procedure** Grp_Nom; **procedure** Grp_Verbal; **public** **constructor** Creer;	**import** java.util.Random; **class** Gener_fr **{** **final** int Maxnbmot=4; **private** String[ ] tnom = **new** String[Maxnbmot]; **private** String[ ] tadjectif = **new** String[Maxnbmot]; **private** String[ ] tarticle = **new** String[Maxnbmot]; **private** String[ ] tverbe = **new** String[Maxnbmot]; **private** String[ ] tadverbe = **new** String[Maxnbmot]; **private** Random ObjAlea = **new** Random**();** **public** Gener_fr**( )** **{** initabl **( );** **}** **private void** initnom **( )** **{** tnom[0] ="chat"; tnom[1] = "chien"; **}** **private void** initadjectif **( )** **{** tadjectif[0] = "beau"; tadjectif[1] = "gentil"; tadjectif[2] = "noir"; tadjectif[3] = "blanc";

```
 procedure phrase;
end;

implementation

 procedure Gener_fr.initnom; begin
  tnom[1]:='chat';
  tnom[2]:='chien';
 end;

 procedure Gener_fr.initverbe; begin
  tverbe[1]:='poursuit';
  tverbe[2]:='aime';
 end;

 procedure Gener_fr.initadverbe; begin
  tadverbe[1]:='malicieusement';
  tadverbe[2]:='joyeusement';
 end;

 procedure Gener_fr.initadjectif; begin
  tadjectif[1]:='beau';
  tadjectif[2]:='gentil';
  tadjectif[3]:='noir';
  tadjectif[4]:='blanc';
 end;

 procedure Gener_fr.initarticle; begin
  tarticle[1]:='le';
  tarticle[2]:='un';
 end;

 procedure Gener_fr.initabl; begin
  Randomize;
  initnom;
  initarticle;
  initverbe;
  initadverbe;
  initadjectif
 end;

 function Gener_fr.Alea(n:integer):integer;
 begin
  Alea:=random(n)+1;
 end;

 procedure Gener_fr.Article; begin
  write(tarticle[Alea(2)],' ')
 end;

 procedure Gener_fr.Nom; begin
  write(tnom[Alea(2)],' ')
 end;

 procedure Gener_fr.Adjectif; begin
  write(tadjectif[Alea(4)],' ')
 end;

 procedure Gener_fr.Adverbe; begin
  write(tadverbe[Alea(2)],' ')
```

```
 }
 private void initadverbe ( )
 {
  tadverbe[0] = "malicieusement";
  tadverbe[1] = "joyeusement";
 }

 private void initverbe ( )
 {
  tverbe[0] = "poursuit";
  tverbe[1] = "aime";
 }
 private void initarticle ( )
 {
  tarticle[0] = "le";
  tarticle[1] = "un";
 }
 private void initabl ( )
 {
  initnom ( );
  initarticle ( );
  initverbe ( );
  initadjectif ( );
  initadverbe ( );
 }
 int Alea(int n)
 {
  return ObjAlea .nextInt(n);
 }
 private void GN  ( )
 {
  if (Alea(2) ==1) // pour  règle 3
  {
   Art ( );
   Nom ( );
   Adj ( );
  }
  else // pour règle 2
  {
   Art ( );
   Adj ( );
   Nom ( );
  }

 }
 private void GV   ( )
 {
  if (Alea(2) ==1) // règle: < verbe >
   Verbe ( );
  else // règle: < verbe > < Adv >
  {
   Verbe ( );
   Adv ( );
  }

 }
 private void Art  ( )
 {
  System.out.print(tarticle[Alea(2)]+' '   ) ;
 }
```

```
end;

procedure Gener_fr.Verbe; begin
 write(tverbe[Alea(2)],' ')
end;

procedure Gener_fr.fin; begin
 writeln('.')
end;

procedure Gener_fr.Grp_Nom; begin
 if Alea(2)=1 then
 begin
  Article;
  Nom;
  Adjectif
 end
 else
 begin
  Article;
  Adjectif;
  Nom
 end

end;
procedure Gener_fr.Grp_Verbal; begin
 if Alea(2)=1 then
  Verbe
 else
 begin
  Verbe;
  Adverbe
 end
end;

procedure Gener_fr.phrase; begin
 Grp_Nom;
 Grp_Verbal;
 Grp_Nom;
 fin
end;

constructor Gener_fr.Creer; begin
 initabl;
end;

end.
```

```
 private void Nom  ( )
 {
  System.out.print(tnom[Alea(2)]+' '   ) ;
 }

 private void Adj  ( )
 {
  System.out.print(tadjectif[Alea(4)]+' '   ) ;
 }

 private void Adv  ( )
 {
  System.out.print(tadverbe[Alea(2)]+' '   ) ;
 }

 private void Verbe  ( )
 {
  System.out.print(tverbe[Alea(2)]+' '   ) ;
 }

 private void fin ( )
 {
  System.out.println('.') ;
 }

 public void phrase( )  // axiome de la grammaire
 {
  GN ( );
  GV ( );
  GN ( );
  fin ( ) ;
 }
}
```

Utiliser la classe Delphi

```
program ProjGenerFr;

 {$APPTYPE CONSOLE}

uses
 SysUtils,
 UclassGenerFr in 'UclassGenerFr.pas';
var
 miniFr:Gener_fr;
 i:integer;
```

Utiliser la classe Java

```
public class UtiliseGenerFr
{
 public static void main(String[] args)
 {
   Gener_fr miniFr = new Gener_fr();
   for(int i=0;i<10;i++)
     miniFr.phrase( );
  }
 }
```

```
begin
 miniFr:=Gener_fr.Creer;
 for i:=1 to 20 do
  miniFr.phrase;
 readln
end.
```

Delphi

Java

```
un chien gentil aime joyeusement le beau chien .
le chien noir aime un chat gentil .
un blanc chien aime le chien gentil .
un beau chat poursuit joyeusement un chien noir .
le chat blanc aime le gentil chat .
un chien beau aime malicieusement un chien gentil .
un noir chat poursuit le beau chien .
un chien blanc aime joyeusement un chien blanc .
le noir chat aime un blanc chien .
un chat noir aime le gentil chat .
un gentil chien aime un blanc chien .
un chat noir aime joyeusement un chien blanc .
le chien blanc aime un blanc chat .
le noir chat poursuit un gentil chat .
un chien blanc aime malicieusement un noir chien .
le blanc chien aime le noir chat .
le gentil chat poursuit le chien gentil .
le chien blanc poursuit joyeusement un chat blanc .
le beau chat aime un chien beau .
un chat blanc aime joyeusement le chien gentil .
```

2. C-grammaires et automates à pile de mémoire

Une C-grammaire est une grammaire de type 2 dans la classification de Chomsky.

Nous adoptons maintenant l'autre point de vue, " **mode analyse** " d'une grammaire, pour s'en servir comme d'un outil de spécification sur la **reconnaissance** des mots du langage engendré par cette grammaire. Cette partie est appelée l'analyse syntaxique.

Dans le cas d'une grammaire de type 3, ce sont les automates d'états finis qui résolvent le problème. Comme ils sont faciles à faire construire par un débutant, nous les avons détaillés dans un paragraphe qui leur est consacré spécifiquement. Dans le cas où G est de type 2 sans être de type 3, nous allons esquisser la solution du problème en utilisant les automates à pile sans fournir de méthodes générales sur leur construction systématique. L'écriture des analyseurs à pile fait partie d'un cours sur la compilation qu'il n'est donc pas question de développer auprès du débutant. Il est toutefois possible de montrer dans le cadre d'une solide initiation sur des exemples bien choisis et simples que l'on peut programmer de tels analyseurs.

Nous retiendrons le côté formateur du principe général de la reconnaissance des mots d'un langage par un ordinateur, aussi bien par les automates d'états finis que par les automates à pile. Nous trouverons aussi une application pratique et intéressante de tels automates dans le filtrage des données. Enfin, lorsque nous élaborerons des interfaces, la reconnaissance de dialogues simples avec l'utilisateur sera une aide à la convivialité de nos logiciels.

2.1 Définition d'un automate à pile

Un automate à pile est caractérisé par la donnée de six éléments :

A = (V_T, E, q0, F, μ , V_p,)
où :
E= ensemble des états (card E est fini)
q0 ∈ E (**q0** , est appelé l'état *initial*).
E ⊂ **F** (F, est appelé l'ensemble des états *finaux*).
V_T =Vocabulaire terminal, contient l'élément #.
V_p = Vocabulaire de la pile, contient toujours 2 éléments spéciaux (notés ω et **Nil**).

ω ∈ V_p (*symbole initial de pile*) **et** **Nil** ∈ V_p

μ : V_T* x E x V_p → E x **Vp' *** (**μ** est appelé *fonction de transition de A*)
avec : **Vp'*** = (**Vp** ∪ { # })* (monoïde sur **Vp** ∪ {#})

Une transition (ou encore règle de transition) a donc la forme suivante :

$$\mu : (a_j , q_i , \alpha) \rightarrow \mu(a_j , q_i , \alpha) = (q_k , x_n)$$

Nous trouvons dans ces automates, une pile (du type **pile LIFO**) dans laquelle l'automate va ranger des symboles pendant son analyse.

2.2 Algorithme de fonctionnement d'un automate à pile

En pratique, afin de simplifier les programmes à écrire, nous définirons et utilisons par la suite un vocabulaire de pile V_p normalisé ainsi :

$V_p = V_p' = V_T \cup \{\#\} \cup \{ \omega , Nil \}$

Intérêt de la notion d'automate :

C'est la fonction de transition qui est l'élément central d'un automate, elle doit être définie de manière à permettre d'**analyser** un mot de V_T*, et aussi de **décider de l'appartenance** ou non d'un mot à un certain langage. Ce langage d'appartenance est appelé le langage reconnu par l'automate.

Nous construisons notre automate à pile comme étant un dispositif physique muni :

- d'une **bande d'entrée** (de papier, ou magnétique par exemple) composée de cases ne pouvant contenir chacune qu'un seul symbole de V_T à la fois,
- d'une autre **bande de pile** composée de cases ne pouvant contenir chacune qu'un seul symbole de V_p à la fois,

- de deux têtes de lecture ou écriture de symboles :

  - l'une de **lecture** capable de reconnaître des éléments du vocabulaire terminal $\mathbf{V_T}$ dans chaque case de la **bande d'entrée**, et **possédant plusieurs états**. La tête de lecture se déplace de gauche à droite d'une case à la fois,

  - l'autre tête de **lecture/écriture** capable de lire ou d'écrire des éléments du vocabulaire de pile $\mathbf{V_p}$ dans chaque case de la **bande de pile**, cette tête se déplace dans les deux sens, mais d'une seule case à la fois.

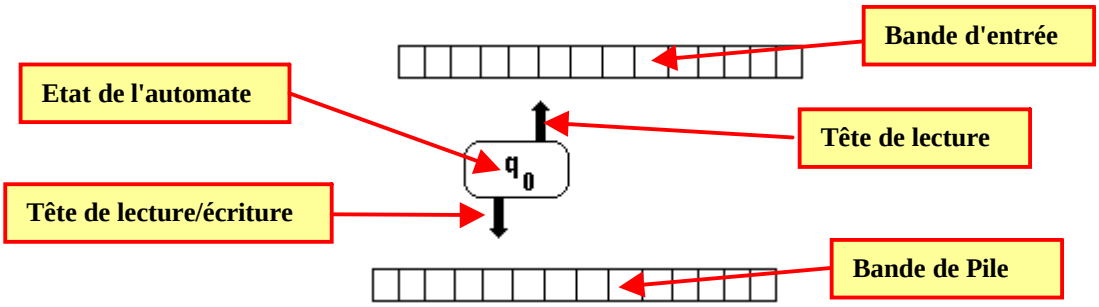
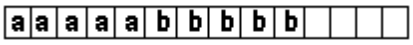


fig - un automate à pile

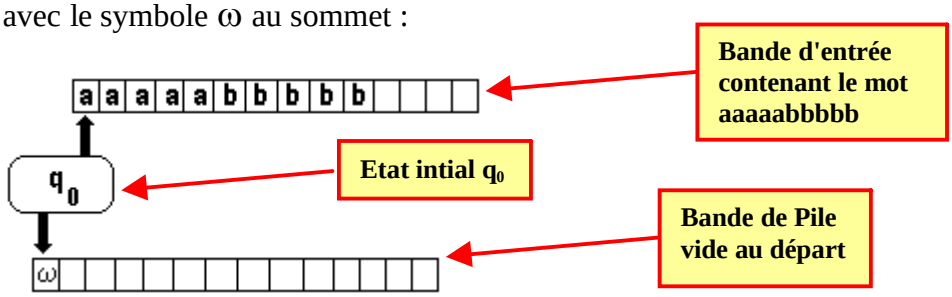
- Les règles de transitions spécifient la manipulation de la bande d'entrée et de la pile de l'automate.

L'algorithme de fonctionnement d'un tel automate est le suivant :

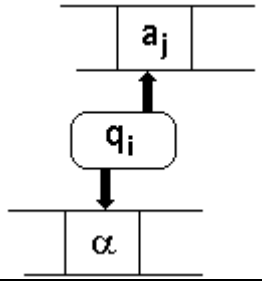
On fournit un mot que l'on écrit symbole par symbole de gauche à droite dans chaque case de l'automate (par exemple avec $\mathbf{V_T}$ ={a,b} le mot **aaaaabbbbb**):



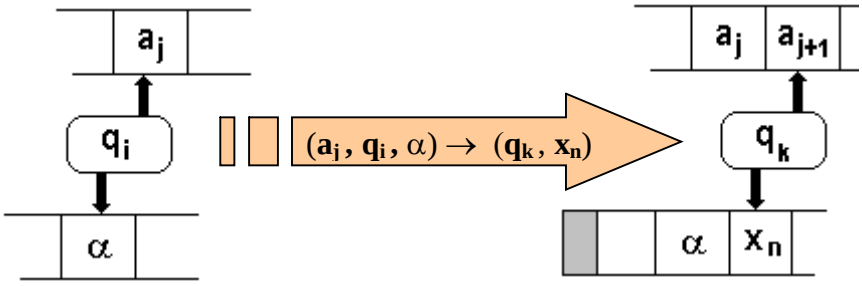
L'automate est mis à l'état initial $\mathbf{q_0}$, sa tête de lecture d'entrée est positionnée sur la première case à gauche de la bande d'entrée(1er symbole du mot à reconnaître), la pile est initialisée avec le symbole ω au sommet :



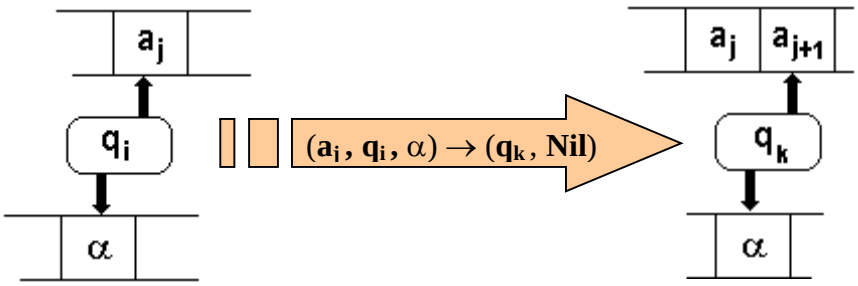
- La tête de lecture se déplace par examen des règles de transition de l'automate en y rajoutant l'examen du sommet de la pile. Le triplet ($\mathbf{a_j}$, $\mathbf{q_i}$,α) enclenche le processus de recherche d'une transition possible dans la partie gauche de la liste des règles de transitions de μ (il y a recherche de la transition μ : ($\mathbf{a_j}$, $\mathbf{q_i}$, α) → .....).

Supposons que la case contienne le symbole $\mathbf{a_j}$ que la tête soit à l'état $\mathbf{q_i}$, et que le sommet de pile ait pour valeur α.	

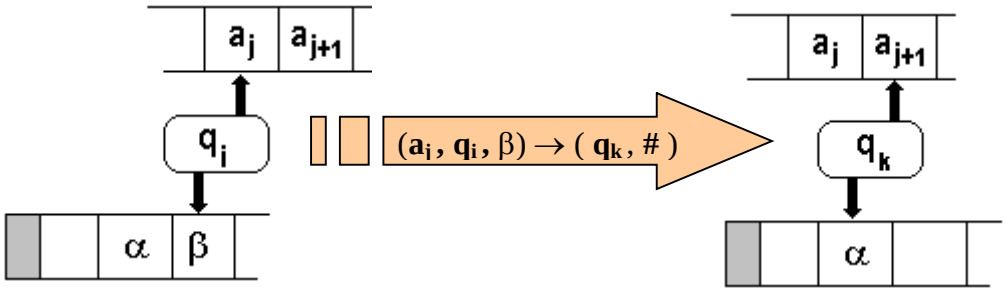
- **La transition** $(\mathbf{a_j}, \mathbf{q_i}, \alpha) \rightarrow (\mathbf{q_k}, \mathbf{x_n})$ signifie que l'automate peut passer de l'état $\mathbf{q_i}$ à l'état $\mathbf{q_k}$ à condition que le mot d'entrée débute par la chaîne préfixe $\mathbf{a_j}$ élément de $\mathbf{V_T}$* (notons que la chaîne $\mathbf{a_j}$ peut être réduite par sa définition à un seul élément de $\mathbf{V_T}$, ce qui est généralement le cas pratique) et que la chaîne α de **Vp** se trouve en sommet de pile. Le résultat de la transition fait que le symbole $\mathbf{a_j}$ est lu et donc reconnu, que la tête d'entrée pointe vers le symbole suivant de la bande d'entrée, que le sommet de la pile a été remplacé par la chaîne $\mathbf{x_n}$ (donc l'élément $\mathbf{x_n}$ a été empilé à la pile), que l'état de l'automate a changé et qu'il vaut maitenant $\mathbf{q_k}$, enfin que la tête de pile pointe sur le nouveau sommet :



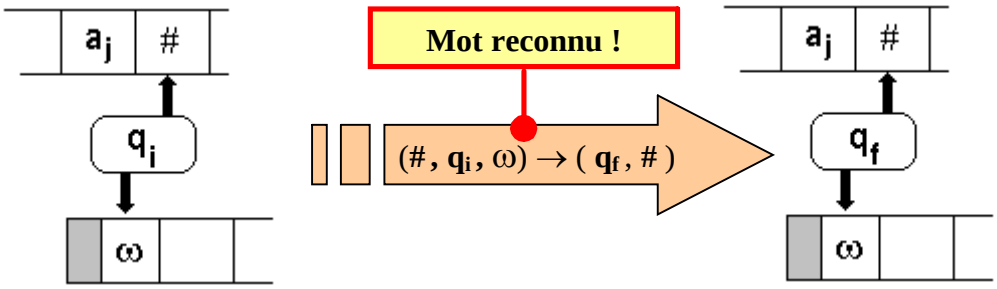
- **La transition** $(\mathbf{a_j}, \mathbf{q_i}, \alpha) \rightarrow (\mathbf{q_k}, \mathbf{Nil})$ signifie pour l'automate de ne rien faire dans la pile. Le résultat de la transition fait que l'état de l'automate passe de $\mathbf{q_i}$ à $\mathbf{q_k}$ et que la tête d'entrée pointe sur le symbole suivant de la bande d'entrée :



- **la transition** $(\mathbf{a_j}, \mathbf{q_i}, \beta) \rightarrow (\mathbf{q_k}, \#)$ signifie effacer l'actuel sommet de pile et pointer sur l'élément d'avant dans la pile (ce qui revient à dépiler la pile). Le résultat de la transition fait que l'état de l'automate passe de $\mathbf{q_i}$ à $\mathbf{q_k}$ et que la tête d'entrée pointe sur le symbole suivant de la bande d'entrée :



- Le mot est reconnu si l'automate rencontre une règle de transition de la forme **(# , q_i , ω) → (q_f , Nil)**, où q_f est un état final. L'automate s'arrête alors.



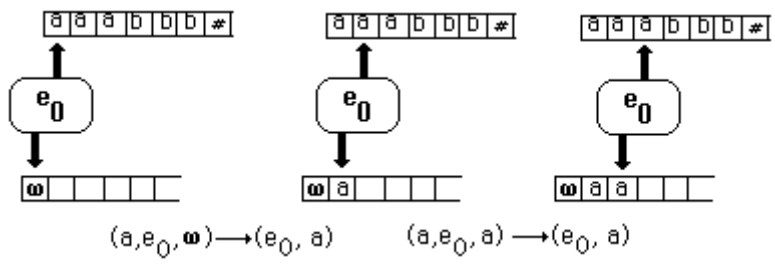
- Si la recherche de la transition μ : **(a_j , q_i ,** α) → .....ne donne pas de résultat on dit que l'auomate bloque : le mot n'est pas reconnu.

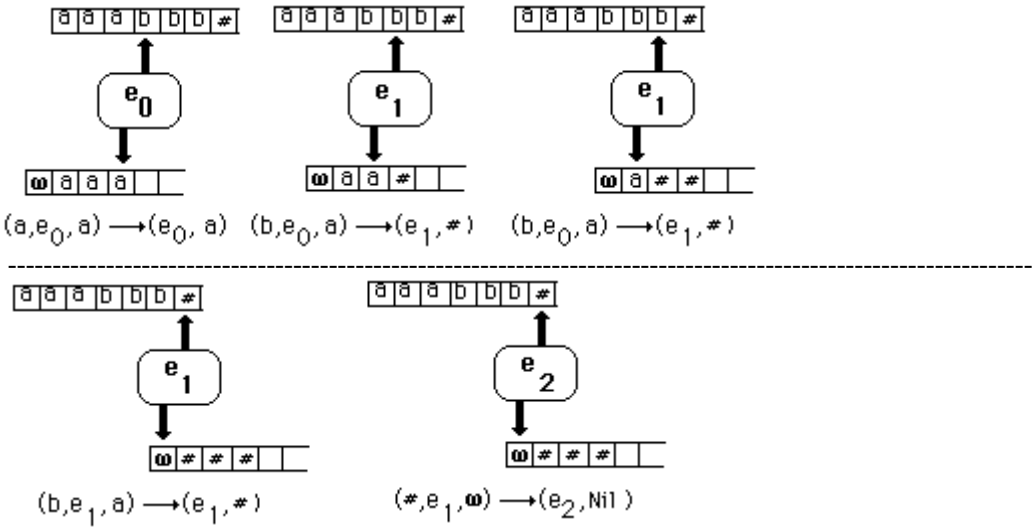
Exemple :

E= {**e0,e1,e2**}
e0 ∈ E (**e0** , état initial)
F ={**e2**} (**F**, état final **e2**)
$\mathbf{V_T}$ = {a,b,#}
$\mathbf{V_p}$ = {a,b,#,ω,**Nil**}

<u>Règles de transitions</u>:
(a , e0, ω **)** → (**e0**, **a**)
(a , e0, a) → (**e0**, **a**)
(b , e0, a) → (**e1**, #)
(b , e1, a) → (**e1**, #)
(# , e1, ω **)** → (**e2**, **Nil**)

Fonctionnement sur un exemple: reconnaissance du mot **aaabbb** :





Propriété :

> Un langage est un C-langage (engendré par une C-grammaire) **ssi** il est accepté par un automate à pile.

L'automate précédent reconnaît le C-langage L suivant :

L = {$\mathbf{a}^n\mathbf{b}^n$}(**a...ab...b**,n symboles **a** concaténés à n symboles **b**, n ≤ 1) sur l'alphabet V_T={**a**,**b**} dont une C-grammaire G est :

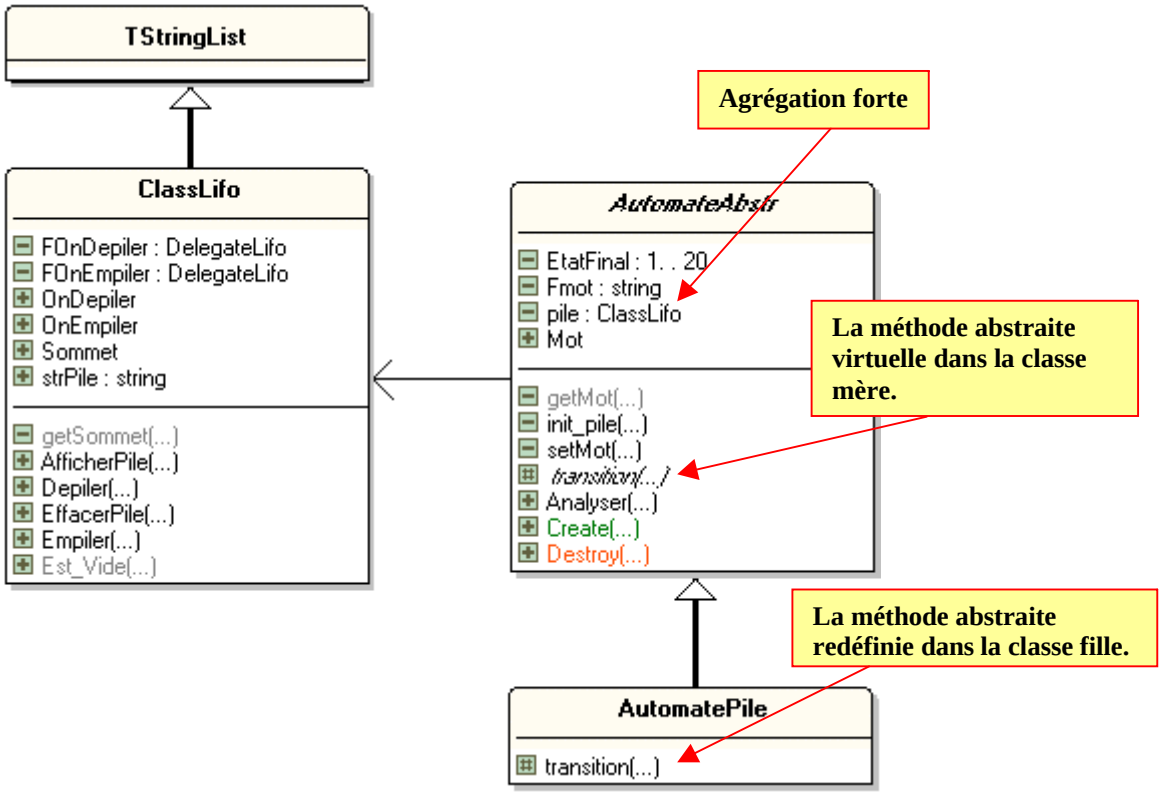
$\mathbf{V_T}$ = { **a** , **b** }
$\mathbf{V_N}$ = { S , A }
Axiome: S
Règles :
1 : S → **aSb**
2 : S → **ab**

2.3 Programme Delphi d'une classe d'automate à pile

Nous utilisons une classe de pile Lifo **ClassLifo** événementielle, déjà définie auparavant pour représenter la pile de l'automate. Afin que la pile Lifo de l'automate gère facilement des éléments de type **string,** au lieu de la faire dériver du type **List** de Delphi, nous proposons de la dériver du type **TStringList** qui est une classe de liste de chaînes :

```
ClassLifo = class (TStringList)
 private
   FOnEmpiler : DelegateLifo ;
   FOnDepiler : DelegateLifo ;
   function getSommet:string;
 public
  strPile:string;
  function  Est_Vide  : boolean ;
  procedure  Empiler (elt : string ) ;
  procedure  Depiler (var elt : string ) ;
  procedure EffacerPile;
  procedure AfficherPile;
  property Sommet:string read getSommet;
  property  OnEmpiler : DelegateLifo  read  FOnEmpiler
            write  FOnEmpiler ;
  property  OnDepiler : DelegateLifo  read  FOnDepiler
            write  FOnDepiler ;
end;
```

Nous construisons une classe abstraite d'automate à pile **AutomateAbstr** qui implante toutes fonctionnalités d'un automate à pile, mais garde abstraite et virtuelle la méthode **transition** qui contient les règles de transitions de l'automate, ceci afin de déléguer son implementation à chaque classe d'automate particulier. Chaque classe fille héritant de **AutomateAbstr** redéfinira la méthode virtuelle **transition**.



Code complet de la classe pile de l'automate

```
unit  ULifoEvent ;

 interface
 uses  classes,Dialogs, SysUtils ;

 type
  DelegateLifo = procedure ( Sender: TObject ; s :string )of object ;

  PileVideException = class (Exception)
end;

ClassLifo = class (TStringList)
 private
  FOnEmpiler : DelegateLifo ;
  FOnDepiler : DelegateLifo ;
  function getSommet:string;
 public
  strPile:string;
  function  Est_Vide  : boolean ;
  procedure  Empiler (elt : string ) ;
  procedure  Depiler (var elt : string ) ;
  procedure EffacerPile;
  procedure AfficherPile;
  property Sommet:string read getSommet;
```

```
   property OnEmpiler : DelegateLifo read FOnEmpiler write FOnEmpiler;
   property OnDepiler : DelegateLifo read FonDepiler write FOnDepiler;
end;

implementation

 procedure ClassLifo.AfficherPile;
 var  i:integer;
 begin
  if self.count<>0 then
   for i:=0 to self.Count-1 do
    write(self.Strings[i],' ');
  writeln
 end;

 procedure  ClassLifo.Depiler(var elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
   elt :=self.Strings[0] ;
   strPile:='';
   self.Delete(0) ;
   if  assigned(FOnDepiler)  then
    FOnDepiler ( self ,elt )
  end
  else
   raise pilevideexception.create ('impossible de dépiler: pile vide' )
 end;

 procedure  ClassLifo.Empiler(elt : string ) ;
 begin
  self.Insert(0 , elt) ;
  if  assigned(FOnEmpiler)  then
   FOnEmpiler ( self ,elt )
 end;

 procedure ClassLifo.EffacerPile;
 begin
  self.Clear;
 end;

 function  ClassLifo.Est_Vide : boolean ;
 begin
  result :=  self.Count = 0 ;
 end;

 function ClassLifo.getSommet: string;
 begin
  result := self.Strings[0]
 end;

end.
```

Code complet des classes AutomateAbstr et AutomatePile

```
unit UAutomPile;

 interface

 uses ULifoEvent;
```

```
 type
  etat=-1..20;
  Vt=char;
  Vp=char;

  AutomateAbstr=class
 private
  EtatFinal:1..20;
  Fmot:string;
  pile:ClassLifo;
  procedure setMot(s:string);
  function getMot:string;
  procedure init_pile;
 protected
  procedure transition(ai:Vt;qj:etat;alpha:Vp; var qk:etat;
                var beta:vp); virtual;  abstract;
 public
  property Mot:string read getmot write setMot;
  procedure Analyser;
  constructor Create(fin:integer);
  destructor Destroy;override;
end;
AutomatePile=class(AutomateAbstr)
 protected
  procedure transition(ai:Vt;qj:etat;alpha:Vp; var qk:etat;
             var beta:vp);  override;
end;

implementation

 {---- AutomateAbstr ----}
const
 omega='$';
 non=-1;
 knil='@';

 constructor AutomateAbstr.Create(fin:integer);
 begin
  pile:=ClassLifo.Create;
  self.init_pile;
  if fin in [1..20] then
   EtatFinal:=fin
  else
   EtatFinal:=20;
  Fmot:='#';
 end;

 destructor AutomateAbstr.Destroy;
 begin
  pile.Free;
  inherited;
 end;

 function AutomateAbstr.getMot: string;
 begin
  result:= Fmot;
 end;

 procedure AutomateAbstr.setMot(s: string);
 var long:integer;
 begin
```

```
  long:=length(s);
  if long<>0 then
  begin
   if s[long]<>'#' then
    Fmot:=s+'#';
  end
  else
   fmot:='#';
end;

procedure AutomateAbstr.init_pile;
begin
 pile.EffacerPile;
 pile.Empiler(omega);
end;

procedure AutomateAbstr.Analyser;
var
 q:etat;
 numcar:integer;
 carPile:Vp;
 s:String;
 begin
  q:=0;
  numcar:=1;
  init_pile;
  while (q<>non)and(q<>EtatFinal) do
  begin
   transition(Fmot[numcar],q,pile.Sommet[1],q,carPile);
   pile.AfficherPile;
   numcar:=numcar+1;
   if carPile='#' then
    pile.Depiler(s)
   else
    if (carpile='a')or(carpile='b') then
     pile.Empiler(carPile);
  end;
  if (q=etatfinal)and(carpile=knil) then
   writeln('chaine reconnue !')
  else
   writeln('blocage, chaine non reconnue !')
 end;

 {---- AutomatePile ----}

procedure AutomatePile.transition(ai:Vt;qj:etat;alpha:Vp;
var qk:etat;
var beta:vp);
begin
 write('(',ai:2,',',qj:2,',',alpha:2,')');
 if (ai='a')and(qj=0)and(alpha=omega) then
 begin
  qk:=0;{ règle ; (a,e0,$) -->(e0,a) }
  beta:='a'
 end
 else
  if (ai='a')and(qj=0)and(alpha='a') then
  begin
   qk:=0; { règle ; (a,e0,a) -->(e0,a) }
   beta:='a'
  end
```

```
    else
     if (ai='b')and(qj=0)and(alpha='a') then
     begin
      qk:=1; { règle ; (b,e0,a) -->(e1,#) }
      beta:='#'
     end
     else
      if (ai='b')and(qj=1)and(alpha='a') then
      begin
       qk:=1; { règle ; (b,e1,a) -->(e1,#) }
       beta:='#'
      end
      else
       if (ai='#')and(qj=1)and(alpha=omega) then
       begin
        qk:=EtatFinal; { règle ; (#,e1,$) -->(e2,Nil) }
        beta:=KNil
       end
       else
       begin {blocage dans tous les autres cas}
        qk:=non;
        beta:=KNil
       end;
  write('--(',qk:2,',',beta,') pile=');
 end;

end.
```

Programme utilisant la classe Automate

```
program ProjAutomPile;

 {$APPTYPE CONSOLE}

 uses
  SysUtils,
  UAutomPile in 'UAutomPile.pas';
 var Automate:AutomatePile;

 begin
  Automate:=AutomatePile.Create(2);
  Automate.Mot:='aaaaabbbbb';
  Automate.Analyser;
  readln;
 end.
```

Exécution de ce programme sur l'exemple ***aaaaabaabb*** *:*

```
( a, 0, $)--( 0,a) pile=$
( a, 0, a)--( 0,a) pile=a $
( a, 0, a)--( 0,a) pile=a a $
( a, 0, a)--( 0,a) pile=a a a $
( a, 0, a)--( 0,a) pile=a a a a $
( b, 0, a)--( 1,#) pile=a a a a a $
( b, 1, a)--( 1,#) pile=a a a a $
( b, 1, a)--( 1,#) pile=a a a $
( b, 1, a)--( 1,#) pile=a a $
( b, 1, a)--( 1,#) pile=a $
( #, 1, $)--( 2,@) pile=$
Chaine reconnue !
```

La construction générale et systématique de tous ces automates à pile dépasse le cadre de ce document, il est conseillé de poursuivre dans les ouvrages signalés dans la bibliographie.

6.2 Automates et grammaires de type 3

Plan du chapitre:

1. Automate pour les grammaires de type 3

1.1 Automates d'états finis déterministes ou non
1.2 Algorithme de fonctionnement d'un AEFD
1.3 Utilisation d'un AEF en reconnaissance de mots
1.4 Graphe d'un automate déterministe ou non
1.5 Matrice de transition : dans le cas déterministe

2. Grammaires et automates

2.1 Automate associé à une K-grammaire
2.2 Grammaire associée à un Automate

3. Implantation d'une classe AEFD en Delphi

3.1 Fonction de transition à partir des règles
3.2 Fonction de transition à partir de la matrice
3.3 Exemple : les identificateurs pascal-like
- Détermination d'une grammaire G_{id} adéquate
- Construction de l'automate associé à G_{id}
- Programme associé à l'automate

3.4 Exemple : les constantes numériques
- Détermination d'une grammaire G_{cte} adéquate
- Construction de l'automate associé à G_{cte}
- Programme associé à l'automate

1. Automates d'états finis pour les grammaires de type 3

Dans ce chapitre, le point de vue adopté est celui de l'implantation pratique des notions proposées en pascal. La reconnaissance automatique et méthodique est très aisément accessible dans le cas des grammaires de type 3 à travers les automates d'états finis. Nous fournissons les éléments théoriques appuyés sur des exemples pratiques.

1.1 Automates d'états finis déterministes ou non

Définition

C'est un Quintuplet $A = (V_T, E, q_0, F, \mu)$ où :
V_T : **vocabulaire terminal** de A.
E : **ensemble des états** de A ; $E = \{q_0, q_1, ..., q_n\}$
$q_0 \in E$ est dénommé **état initial** de A.
$F \subset E$: F est l'ensemble des **états finaux** de A.
$\mu : E \times V_T \rightarrow E$, une **fonction de transition** de A.

Définition

Un automate A, $A = (V_T, E, q_0, F, \mu)$, est dit **déterministe**, si sa fonction de transition μ est une vraie fonction au sens mathématique. Ce qui revient à dire qu'un couple de $E \times V_T$ n'a qu'**une seule image** par μ dans E.

Intérêt de la notion d'automate d'états finis

Comme pour les automates à pile, c'est la fonction de transition qui est l'élément central de l'automate A. Elle doit être définie de manière à permettre d'analyser un mot de V_T^*, et aussi de décider de l'appartenance ou non d'un mot à un certain langage. Ce langage d'appartenance est appelé le langage reconnu par l'automate.

Exemple :

Soit un automate possédant parmi ses règles, les trois suivantes : $(q_i, a_j) \rightarrow q_k^1$ $(q_i, a_j) \rightarrow q_k^2$ ..... $(q_i, a_j) \rightarrow q_k^n$	Il existe trois règles ayant la même partie gauche (q_i , a_j), ce qui revient à dire que le couple (q_i , a_j) a trois images distinctes, donc l'automate est **non déterministe**.

Par la suite, pour des raisons de simplification pratique, nous considérerons les **A**utomates d'**E**tats **F**inis normalisés que nous nommerons AEF, en posant :

F = { q_f } un seul état final
$V_T = V_T \cup \{ \# \}$ on ajoute dans V_T un symbole terminal de fin de mot #, le même pour tous les AEF.

1.2 Fonctionnement pratique d'un AEF :

Nous construisons notre AEF comme étant un dispositif physique muni :

- d'une **bande d'entrée** (de papier, ou magnétique par exemple) composée de cases ne pouvant contenir chacune qu'un seul symbole de $\mathbf{V_T}$ à la fois,

- d'une seule tête de lecture de symobles capable de reconnaître des éléments du vocabulaire terminal $\mathbf{V_T}$ dans chaque case de la **bande d'entrée**, et **possédant plusieurs états**. La tête de lecture se déplace de gauche à droite d'une case à la fois,

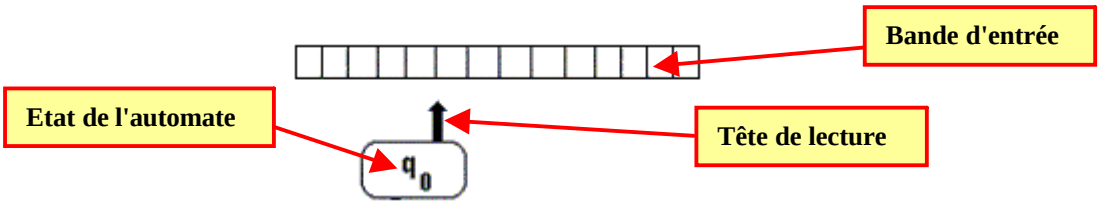
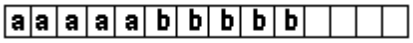


fig - un automate d'états finis

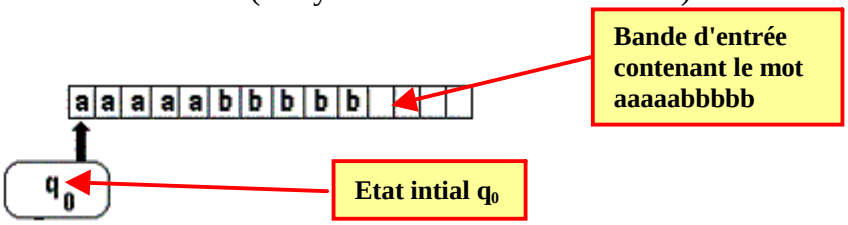
- Les règles de transitions spécifient la manipulation de la bande d'entrée de l'automate.

L'algorithme de fonctionnement d'un AEF :

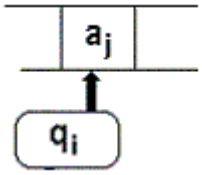
L'algorithme est très semblable à celui d'un automate à pile (en fait on peut considérer qu'il s'agit d'un cas particulier d'automate à pile dans lequel on n'effectue jamais d'action dans la pile), on fournit un mot que l'on écrit symbole par symbole de gauche à droite dans chaque case de l'automate (par exemple avec $\mathbf{V_T}$ ={a,b} le mot **aaaaabbbbb**):



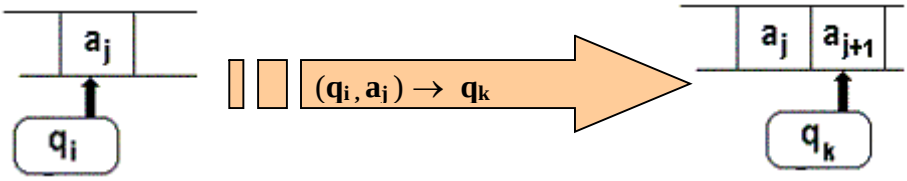
L'automate est mis à l'état initial $\mathbf{q_0}$, sa tête de lecture d'entrée est positionnée sur la première case à gauche de la bande d'entrée(1er symbole du mot à reconnaître) :



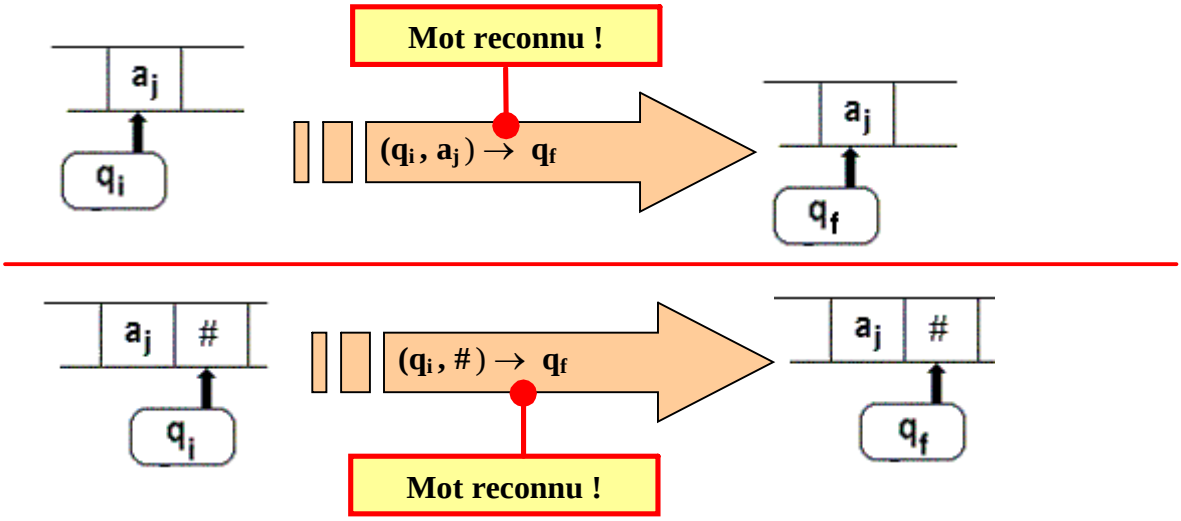
- La tête de lecture se déplace par examen des règles de transition de. Le couple $(\mathbf{a_j}, \mathbf{q_i})$ enclenche le processus de recherche d'une transition possible dans la partie gauche de la liste des règles de transitions de μ (il y a recherche de la transition μ : $(\mathbf{a_j}, \mathbf{q_i}) \rightarrow$ .....).

Supposons que la case contienne le symbole **a_j** que la tête soit à l'état **q_i**	

- **La transition (q_i , a_j) → q_k** signifie que l'automate peut passer de l'état **q_i** à l'état **q_k** à condition que le mot d'entrée débute par la chaîne préfixe **a_j** élément de **V_T*** (notons que la chaîne **a_j** peut être réduite par sa définition à un seul élément de **V_T** , ce qui est généralement le cas pratique. Le résultat de la transition fait que le symbole **a_j** est lu et donc reconnu, que la tête d'entrée pointe vers le symbole suivant de la bande d'entrée :



- Le mot est reconnu si l'automate rencontre une règle de transition de la forme **(q_i , a_j) → q_f** ou bien **(q_i , #) → q_f** où **q_f** est un état final. L'automate s'arrête alors.



- Si l'AEF ne trouve pas de règle de transition commençant par (q_j , a_i), c'est à dire que le couple (q_j , a_i) n'a pas d'image par la fonction de transition μ, on dit alors que l'automate **bloque** : le mot n'est pas reconnu comme appartenant au langage.

1.3 Utilisation d'un AEF en reconnaissance de mots

Soit la grammaire G_1 déjà étudiée précédemment dont le langage engendré est celui des mots de la forme $a^n b^p$.

grammaire G_1	Soit l'automate A_1 :
$L(G_1) = \{ a^n b^p / (n \leq 1) \text{ et } (p \leq 1) \}$ $G_1 : V_N = \{S,A\}$ $V_T = \{a,b\}$ **Axiome** : S **Règles** 1 : $S \rightarrow aS$ 2 : $S \rightarrow aA$ 3 : $A \rightarrow bA$ 4 : $A \rightarrow b$	$V_T = \{a,b\}$ $E = \{q_0 , q_1 , qf \}$ $\mu : (q_0, a) \rightarrow q_0$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_f$ *(A_1 est non déterministe)*

*Fonctionnement pratique de l'AEF A_1 sur le mot a^3b^2 (**aaabb**) :*

a a a b b ... q_0 $(q_0,a) \longrightarrow^1 q_0 \Rightarrow$	a a a b b ... q_0 $(q_0,a) \longrightarrow^1 q_0 \Rightarrow$
a a a b b ... q_0 $(q_0,a) \longrightarrow^2 q_1 \Rightarrow$	a a a b b ... q_1 $(q_1, b) \longrightarrow^3 q_1 \Rightarrow$
a a a b b ... q_1	$(q_1 , b) \rightarrow^4 q_f$ règle 4 $\Rightarrow$ l'AEF s'arrête, le mot **aaabb** est reconnu !

Nous remarquons que dans le cas de cet AEF, il nous a fallu aller " voir " un symbole plus loin, afin de déterminer la bonne règle de transition pour mener jusqu'au bout l'analyse.

On peut montrer que tout AEF non déterministe peut se ramener à un AEF déterministe équivalent. Nous admettons ce résultat et c'est pourquoi nous ne considérerons par la suite que les AEF déterministes (notés **AEFD**).

Voici à titre d'exemple un AEFD équivalent à l'AEF A_1 précédent :

Soit l'AEFD A_2 reconnaissant le langage $L=\{ a^n b^p / (n \leq 1) \text{ et } (p \leq 1) \}$, nous aurons donc deux automates reconnaissant le langage L(l'un est déterministe, l'autre est non déterministe), ci-dessous un tableau comparatif de ces deux automates d'états finis :

AEF A_1 non déterministe	AEF A_2 déterministe
$V_T = \{a,b\}$ $E = \{q_0 , q_1 , qf \}$ $\mu : (q_0, a) \rightarrow q_0$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_1, b) \rightarrow q_1$	$V_T = \{a,b,\#\}$ $E = \{q_0,q_1, q_2,qf \}$ $\mu : (q_0, a) \rightarrow q_1$ $\mu : (q_2, b) \rightarrow q_2$ $\mu : (q_1, a) \rightarrow q_1$

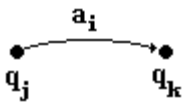
$\mu : (q_1, b) \rightarrow q_f$	$\mu : (q_1, b) \rightarrow q_2$ $\mu : (q_2, \#) \rightarrow q_f$

Voici la reconnaissance automatique du mot $\mathbf{a^3b^2}$ par l'automate AEFD A_2 :
$(q_0, \mathbf{a}) \rightarrow q_1$
$(q_1, \mathbf{a}) \rightarrow q_1$
$(q_1, \mathbf{a}) \rightarrow q_1$
$(q_1, \mathbf{b}) \rightarrow q_2$
$(q_2, \mathbf{b}) \rightarrow q_2$
$(q_2, \#) \rightarrow q_f$ mot reconnu !

1.4 Graphe d'un automate déterministe ou non

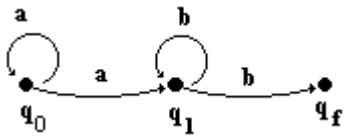
C'est un graphe orienté, représentant la suite des transitions de l'automate comme suit :

$(q_j , a_i) \rightarrow q_k$ est représentée par l'arc à 2 sommets :

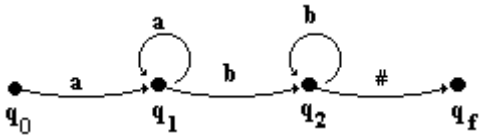


Exemples du graphe des deux AEF précédents :

graphe de A_1 :



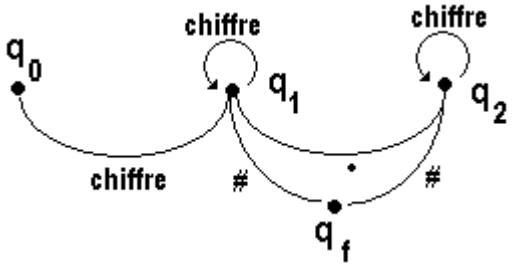
graphe de A_2 :



Que l'AEF soit déterministe ou non, il est toujours possible de lui associer un tel graphe.

Exemple : reconnaître des chiffres

Voici un **A**utomate d'**E**tats **F**inis **D**éterministe qui reconnaît :
- les constantes chiffrées terminées par un #,

AEFD représenté par son graphe	AEFD représenté par ses règles
	$(q_0 , \mathbf{chiffre}) \rightarrow q_1$ $(q_1, \mathbf{chiffre}) \rightarrow q_1$ $(q_1 , \#) \rightarrow q_f$ $(q_1 , .) \rightarrow q_2$ $(q_2 , \mathbf{chiffre}) \rightarrow q_2$ $(q_2 , \#) \rightarrow q_f$ où $(q_0 , \mathbf{chiffre})$ représente un notation pour les 10 couples : $(q_0 , \mathbf{0}),\ldots, (q_0 , \mathbf{9})$

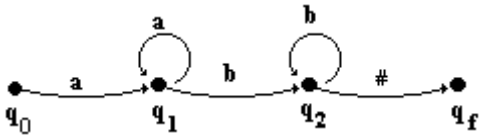
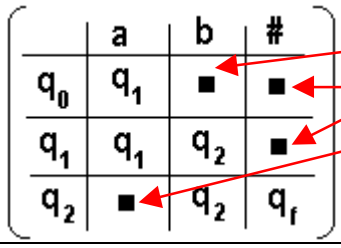
On peut voir à travers ce dernier exemple, le schéma général d'un analyseur lexical qui constitue la première étape d'un compilateur. Il suffit dès que le symbole a été reconnu donc à partir d'un état final $\mathbf{q_f}$ de repartir à l'état q_0.

1.5 Matrice de transition : dans le cas déterministe

On représente la fonction de transition par une matrice **M** dont les cellules sont toutes des états de l'AEF (ensemble E), où les colonnes contiennent les éléments de V_T (symboles terminaux), les lignes contiennent les états (éléments de E sauf l'état final q_f).

La règle (q_j , a_i) $\rightarrow$ q_k est stockée de la façon suivante M(i,j)= q_k où :
la ligne j correspond à l'état q_j
la colonne i correspond au symbole a_i

Exemple : la matrice des transitions de A_2

AEFD A2 représenté par son graphe	AEFD A2 représenté par sa matrice
	

	a	b	#
q_0	q_1	■	■
q_1	q_1	q_2	■
q_2	■	q_2	q_f

Il n'y a pas d'image pour la fonction de transition , donc blocage de A2.

Utilisation pratique de la matrice des transitions

Dénommons Mat[i,j] l'état valide de coordonnées (i,j) dans la matrice des transitions d'un AEFD. Un schéma d'algorithme de reconnaissance par l'AEFD est très simple à décrire :

```
Etat ←q0 ;
Symlu ← premier symbole du mot ;
tantque Etat ≠ qf Faire
 Etat ←Mat[Etat,Symlu] ;
 Symlu ←Symbole suivant
Fintant ;
```

2. Automates et grammaires de type 3

Il existe une correspondance bijective entre les K-grammaires (**grammaires de type-3**) et les **AEF**. Cette correspondance est la base sur laquelle nous systématisons l'implantation d'un AEFD. En voici une construction pratique.

2.1 Automate associé à une K-grammaire

Soit G une K-grammaire , $G = (V_N, V_T, S, R)$
On cherche l'AEF A, $A = (V_T', E, q_0, q_f, \mu)$, tel que A reconnaisse G.

Soit la construction suivante de l'AEF A :

Grammaire G	AEF A associé
V_T	$V_T' = V_T \cup \{\#\}$
Chaque élément de V_N est un q_j de E	$E = V_N \cup \{q_f\}$
A toute règle terminale de G de la forme $A_j \rightarrow a_k$	on associe la règle de transition $(q_j, a_k) \rightarrow q_f$
S est l'axiome de G	$q_0 = S$
A toute règle non terminale de G de la forme $A_j \rightarrow a_k A_i$	on associe la règle de transition $(q_j, a_k) \rightarrow q_i$

L'automate A ainsi construit reconnaît le langage engendré par G.

Remarque :

L'automate A_1 reconnaissant le langage $\{a^n b^p / (\mathbf{n} \leq 1) et (\mathbf{p} \leq 1)\}$ associé à la grammaire G_1 (cf. plus haut) a été construit selon cette méthode.

Exemple : soit G une K-grammaire suivante et **Aut** l'automate associé par le procédé bijectif précédent

G K-grammaire	**Aut** automate associé
$G = (V_N, V_T, S, R)$ $V_T = \{ a, b, c, \# \}$ $V_N = \{ S, A, B \}$ **Axiome** : S **Règles** de G : 1 : $S \rightarrow aS$ 2 : $S \rightarrow bA$ 3 : $A \rightarrow bB$ 4 : $B \rightarrow cB$ 5 : $B \rightarrow \#$	**Aut** $= (V_T', E, q_0, q_f, \mu)$ $V_T' = V_T$ S est associé à : q_0 A est associé à : q_1 B est associé à : q_f 1 : $S \rightarrow aS$ est associé à :$(q_0, a) \rightarrow q_0$ 2 : $S \rightarrow bA$ est associé à :$(q_0, b) \rightarrow q_1$ 3 : $A \rightarrow bB$ est associé à :$(q_1, b) \rightarrow q_2$ 4 : $B \rightarrow cB$ est associé à :$(q_2, c) \rightarrow q_2$ 5 : $B \rightarrow \#$ est associé à :$(q_2, \#) \rightarrow q_f$

Etudions maintenant la construction réciproque d'une K-grammaire à partir d'un AEF.

2.2 Grammaire associée à un Automate

Soit l'AEF **Aut**, tel que A = (V_T' , E , q_0 , q_f ,μ)
On cherche G = (V_N,V_T,S,R) une K-grammaire du langage reconnu par cet automate.

AEF A ut	Grammaire G associée
V_T'	$V_T = V_T'$
E	$V_N = E - \{q_f\}$
q_0	**Axiome** : q_0
si $q_j \neq q_f$ **alors** pour $(q_j, a_k) \rightarrow q_i$	on construit : [r : $q_j \rightarrow a_k q_i$] dans G
si $q_j = q_f$ **alors** pour $(q_j, a_k) \rightarrow q_f$	on construit : [r : $q_j \rightarrow a_k$] dans G

Exemple : soit l'automate Aut reconnaissant le langage $\{a^n b^2 c^p \ / \ n \leq 0 \text{ et } p \leq 0\}$ et G une grammaire associée

Aut automate	G K-grammaire associée
V_T = { a, b, c, # } E = {q_0,q_1, q_2,qf } **transitions :** (1) (q_0, a) → q_0 *[les a^n]* (2) (q_0, b) → q_1 (3) (q_1, b) → q_2 *[le b^2]* (4) (q_2, c) → q_2 *[les c^p]* (5) (q_2, #) → q_f	S remplace q_0 On pose : V_T = { a, b, c, # } A remplace q_1 On pose : V_N = { S, A, B } B remplace q_2 **Axiome** : S **règles :** 1 : S → aS remplace (q_0, a) → q_0 2 : S → bA remplace (q_0, b) → q_1 3: A → bB remplace (q_1, b) → q_2 4 : B → cB remplace (q_2, c) → q_2 5 : B → # remplace (q_2, #) → q_f

La grammaire **G** de type 3 associée par la méthode précédente est celle que nous avons déjà vue dans l'exemple précédent.

1. Implantation d'une classe d'AEFD en Delphi

Nous proposons deux constructions différentes de la fonction de transition d'un AEFD soit en utilisant directement les règles de transition, soit à partir de la matrice de transition.

3.1 Fonction de transition à partir des règles

Méthodologie

Toute règle est de la forme $(q_j, a_k) \rightarrow q_i$, donc nous pourrons prendre comme modèle d'implantation de la fonction de transition une **méthode function** d'une classe que nous nommerons AutomateEF, dont voici ci-dessous une écriture fictive pour une seule règle.

```
function AutomateEF.transition(q:T_etat;car:Vt):T_etat ;
begin
  if (q= qj)and(car= ak) then q:= qi {la règle (qj, ak) → qi}
  else ....
end ;
```

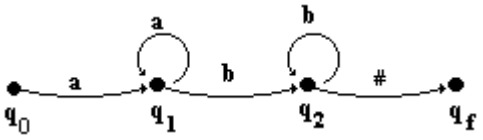
Toutes les autres règles sont implantées dans la **méthode** transition de la même façon.

L'appel de la **méthode** transition se fera à travers un objet AEFD de classe AutomateEF :

EtatApres := AEFD.transition(q_j, a_k) ; ***{la fonction renvoie l'état q_i}***

Exemple : morceaux de code

la méthode transition de l'automate déterministe A_2 :
(reconnaissant le langage $a^n b^p$)




```
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
....
AutomateEF = class
  q : T_etat;
  mot:string;
  function transition(q:T_etat;car:Vt):T_etat;
end;
```

```
Implementation

function AutomateEF.transition(q:T_etat;car:Vt):T_etat;
{par les règles de transition :}
begin
  if (q=0)and(car='a') then q:=1 {(q0,a) → q1}
  else if (q=1)and(car='a') then q:=1 {(q1,a) → q1}
  else if (q=1)and(car='b') then q:=2 {(q1,b) → q2}
  else if (q=2)and(car='b') then q:=2 {(q2,b) → q2}
  else if (q=2)and(car=finmot) then q:=fin {(q2,#) → qf}
  else q:=imposs; {blocage, le caractère n'est pas dans Vt}
  result :=q
end;
```

Appel de la méthode transition dans le programme principal :

```
{Analyse}

numcar:=1;
etat:=0;
while (etat<>imposs)and(etat<>fin) do
begin
  Symsuiv(numcar,symlu); {fournit dans symlu le symbole suivant}
  etat:= AEFD.transition(etat,symlu);
end;
```

Comme l'automate est déterministe, il est possible de procéder différemment en utilisant sa matrice de transition, ce qui est un bon exemple d'application d'utilisation de la notion de matrice dans un programme.

3.2 Fonction de transition à partir de la matrice

Méthodologie

Une autre écriture d'un même AEFD est obtenue (lorsque cela est possible en place mémoire) à partir d'un **tableau** Delphi (dénoté **table**) représentant la matrice des transitions de l'automate. Nous utilisons le même modèle d'implantation de la fonction de transition que dans le cas d'une description par règles (**méthode function** d'une classe AutomateEF).

Version réduite initiale du code : morceaux de code

```
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
 T_transition=array[T_etat,char] of T_etat;
```

```
AutomateEF = class

 q : T_etat;
 mot:string;
 table:T_transition; {matrice des transitions}

end;
```

Il est nécessaire d'initialiser la matrice table avec les valeurs de départ de l'AEFD. Une méthode **init_table** pourra se charger de ce travail. Dans ce cas, la **méthode transition** est très simple à écrire, elle se résume à parcourir la matrice des transitions accessible comme champ de la classe :

Version augmentée du code : morceaux de code

```
AutomateEF = class
  q : T_etat;
  mot:string;
  table:T_transition; {matrice des transitions}
  procedure init_table;
  function transition(q:T_etat;car:Vt):T_etat;
end;

implementation

procedure AutomateEF.init_table;
{initialisation de la table des transitions}

var
 i:T_etat;
 j:0..255;
 k:char;
```

```
begin {init_table}
  for i:=imposs to fin do
    for j:=0 to 255 do
        Chargementde(table)
   ......
end; {init_table}

function AutomateEF.transition(q:T_etat;car:Vt):T_etat;
{par la table de transition :}
begin
  q := table[q,car];
  result := q;
end;
```

L'appel de la **méthode** transition se fait comme dans le cas précédent, à travers un objet AEFD de classe AutomateEF :

EtatApres := AEFD.transition(q_j, a_k) ; ***{la fonction renvoie l'état q_i}***

Exemple : le même automate (reconnaissant le langage $a^n b^p$)

Nous reprenons l'automate du paragraphe précédent, mais en l'implantant grâce à sa table de transition.

morceaux de code

la méthode transition de l'automate déterministe A_2 :
(reconnaissant le langage $a^n b^p$)

	a	b	#
q_0	q_1	■	■
q_1	q_1	q_2	■
q_2	■	q_2	q_f

```
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
....
AutomateEF= class
 private
   EtatFinal : T_etat;
   Fmot:string;
   table:T_transition; {matrice des transitions}
  procedure init_table;
  function transition(q:T_etat;car:Vt):T_etat;
end;

Implementation

procedure AutomateEF.init_table;
{initialisation de la table des transitions}
var
 i:T_etat;
 j:0..255;
 k:char;
```

```
 begin {init_table}
   for i:=imposs to fin do
    for j:=0 to 255 do
      table[i,chr(j)]:=imposs;
   {par défaut tout est non reconnu}
    table[0,'a']:=1;
    table[1,'a']:=1;
    table[1,'b']:=2;
    table[2,'b']:=2;
    table[2,finmot]:=EtatFinal
end; {init_table}


 function AutomateEF.transition(q:T_etat;car:Vt):T_etat;
 {par la table de transition :}
 begin
   q := table[q,car];
   result := q;
 end;
```

Appel de la méthode transition dans le programme principal :

```
{Analyse}

numcar:=1;
etat:=0;
while (etat<>imposs)and(etat<>fin) do
begin
  Symsuiv(numcar,symlu); {fournit dans symlu le symbole suivant}
  etat:= AEFD.transition(etat,symlu);
end;
```

La unit Delphi contenant une classe abstraite d'automate et une classe fille d'AEF Reconnaissant le langage $a^n b^p$

```
Unit UautomEF;

interface
const
 imposs=-1;
 fin=20;
 finmot='#';
type
 T_etat=imposs..fin;
 Vt=char;
 T_transition=array[T_etat,char] of T_etat;

AutomateAbstr = class
 private
   Fmot:string;
   table:T_transition; {matrice des transitions}
   procedure setMot(s:string);
   function getMot:string;
   function transition(q:T_etat;car:Vt):T_etat;
   procedure Symsuiv (n: integer; var car:Vt);
protected
    EtatFinal : T_etat;
   procedure init_table; virtual;  abstract;
public
  property Mot:string read getmot write setMot;
  procedure Analyser;
  constructor Create(fin:integer);
end;

AutomateEF=class(AutomateAbstr)
 protected
    procedure init_table; override;
end;

Implementation
{--- AutomateAbstr ---}
constructor AutomateAbstr.Create(fin:integer);
 begin
    if fin in [1..20] then
     EtatFinal:=fin
   else
     EtatFinal:=20;
   Fmot:='#';
  init_table;
 end;

 function AutomateAbstr.getMot: string;
 begin
    result := Fmot;
 end;

procedure Symsuiv (n: integer; var car:Vt);
begin
 car := Fmot[n];
end;
```

```
procedure AutomateAbstr.setMot(s: string);
 var long : integer;
 begin
   long:=length(s);
   if long<>0 then   begin
     if s[long]<>'#' then
       Fmot:=s+'#';
   end
   else
    Fmot := '#';
 end;

function AutomateAbstr.transition(q:T_etat;car:Vt):T_etat;
   {par la table de transition :}
begin
     q := table[q,car];
     result := q;
end;

procedure AutomateAbstr.Analyser;
var
 etat : T_etat;  numcar : integer;
  s : string;  symlu : Vt;
begin
  numcar:=1;
  etat:=0;
  while (etat<>imposs)and(etat<> EtatFinal) do
  begin
     Symsuiv (numcar , symlu);
     numcar:= numcar+1;
    {fournit dans symlu le symbole suivant}
    etat:= self.transition(etat,symlu);
  end;
  if etat =etatfinal then
    writeln('chaine reconnue !')
  else
   writeln('blocage, chaine non reconnue !')
end;
{--- AutomateEF ---}
 procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  k:char;
 begin
  for i:=imposs to fin do
        for j:=0 to 255 do
          table[i,chr(j)]:=imposs;
         {par défaut tout est non reconnu}
   table[0,'a']:=1;
   table[1,'a']:=1;
   table[1,'b']:=2;
   table[2,'b']:=2;
   table[2,finmot]:= EtatFinal;
 end;
end.
```

Programme utilisant la classe AutomateEF

Reconnaissant le langage L = { $a^n b^p$ / (n ≤ 1) et (p ≤ 1) }

```
program ProjAutomEF;

 {$APPTYPE CONSOLE}

 uses
  SysUtils,
  UAutomEF in 'UAutomEF.pas';
 var AEFD:AutomateEF;

  begin
    AEFD:= AutomateEF.Create(3);
    AEFD.Mot:='aaaaabbbbb';
    AEFD.Analyser;
    readln;
  end.
```

Exécution de ce programme sur l'exemple ***aaaaabbbbb*** *:*

```
( 0, a)--> 1
( 1, a)--> 1
( 1, a)--> 1
( 1, a)--> 1
( 1, a)--> 1
( 1, b)--> 2
( 2, b)--> 2
( 2, b)--> 2
( 2, b)--> 2
( 2, b)--> 2
( 2, #)--> 3
chaine reconnue !
```

Exécution de ce programme sur l'exemple ***aaaaabbabb*** *:*

```
( 0, a)--> 1
( 1, a)--> 1
( 1, a)--> 1
( 1, a)--> 1
( 1, a)--> 1
( 1, b)--> 2
( 2, b)--> 2
( 2, a)--> -1
blocage, chaine non reconnue !
```

Nous remarquons dans ce dernier paragraphe, que nous avons mis en place un procédé général qui permet de construire méthodiquement en Delphi une classe d'AEFD, **uniquement** en changeant le contenu de la méthode **init_table**. Nous avons en fait mis au point un générateur manuel de programme Delphi pour AEFD.

Par la suite, nous utiliserons ce procédé à chaque fois que nous aurons à programmer un AEFD. Nous n'aurons seulement qu'à déclarer une nouvelle classe héritant de AutomateAbstr et implémenter par redéfinition sa méthode **init_table** :

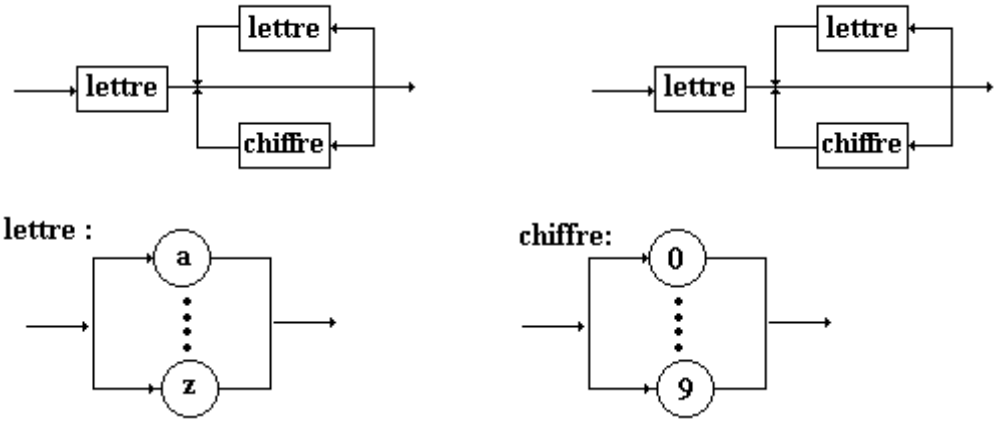
```
AutomateEF = class ( AutomateAbstr )
 protected
   procedure init_table; override;
end;

Implementation
```

```
(* --- AutomateEF
      Reconnaissant le langage L = { a^n b^p / (n ≤ 1) et (p ≤ 1) }
---*)
procedure AutomateEF.init_table;
{initialisation de la table des transitions}
var  i:T_etat;  j:0..255;  k:char;
begin
  for i:=imposs to fin do
      for j:=0 to 255 do
        table[i,chr(j)]:=imposs;
       {par défaut tout est non reconnu}
   table[0,'a']:=1;
   table[1,'a']:=1;
   table[1,'b']:=2;
   table[2,'b']:=2;
   table[2,finmot]:= EtatFinal;
end;
```

Nous terminons ce chapitre en détaillant complètement deux exercices de construction de programmes selon la méthode qui vient d'être décrite. Le lecteur pourra en inventer d'autres basés sur la même idée.

3.3 Exemple : les identificateurs pascal-like

On donne des diagrammes syntaxiques de construction des identificateurs Pascal :
identificateur :



Construisons un programme Delphi reconnaissant de tels identificateurs, en utilisant le procédé de génération manuelle avec la classe abstraite AutomateAbstr définie au paragraphe précédent.

Méthode de travail adoptée

- **Détermination d'une grammaire G adéquate.**
- **Construction de l'automate AEFD associé à G.**
- **Programme Delphi associé à l'automate construit.**

Détermination d'une grammaire G_{id} adéquate

Nous remarquons d'abord qu'il y a une grammaire de type 3 engendrant ces identificateurs :
Soient à considérer les ensembles :
Lettre = { **a**, **b**,..., **z** }. *// les 26 lettres de l'alphabet*
Chiffre = { **0**, **1**,..., **9** }. *// les 10 chiffres*
V_T = Chiffre ∪ Lettre ∪ {#}.
V_N = { ⟨*identificateur*⟩ , **A** }
Posons G_{id} = (V_T , V_N , ⟨*identificateur*⟩ , Règles) une grammaire où
<u>Axiome</u> : ⟨*identificateur*⟩
<u>Règles</u> :
⟨*identificateur*⟩ → **a** |**b**|...|**z** **A**
A → **a** |**b**|...|**z** **A**
A → **0** |**1**|...|**9** **A**
A → #
La grammaire G_{id} est de type 3 déterministe.

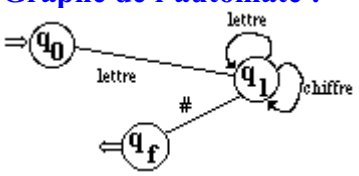
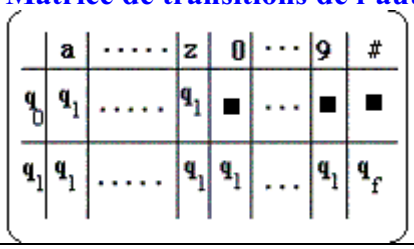
Construction de l'automate associé à G_{id}

Afin de réduire le nombre de lignes de texte, nous adoptons les conventions d'écriture suivantes :

(q_k ,Lettre) → q_i , *représente l'ensemble des 26 règles :*
(q_k ,**a**) → q_i
.....
(q_k ,**z**) → q_i

(q_k , Chiffre) → q_i , *représente l'ensemble des 10 règles :*
(q_k ,**0**) → q_i
.....
(q_k ,**9**) → q_i

Etats associés aux éléments de V_N: ⟨*identificateur*⟩ ⇔ q_0 ⟨A⟩ ⇔ q_1 Etat initial = q_0 Etat final = q_f **Vocabulaire terminal de l'automate :** $V_T' = V_T \cup \{\#\}$ = Chiffre ∪ Lettre ∪ {#}	**Règles de l'automate associé à G_{id} :** (q_0 , Lettre) → q_1 *// 26 règles* (q_1 , Lettre) → q_1 *// 26 règles* (q_1 , Chiffre) → q_1 *// 10 règles* (q_1 , #) → q_f Soit un total de 63 règles.

Graphe de l'automate : 	**Matrice de transitions de l'automate:** 

	a	.....	z	0	...	9	#
q_0	q_1	.....	q_1	■	...	■	■
q_1	q_1	.....	q_1	q_1	...	q_1	q_f

Programme associé à l'automate

Nous héritons de la classe AutomateAbstr. Nous n'avons que la méthode **init_table** à redéfinir

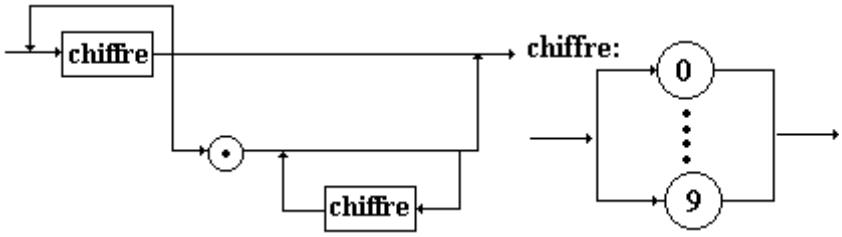
```
AutomateEF = class ( AutomateAbstr )
 protected
   procedure init_table; override;
end;

implementation

procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  x:char;
 begin
   for i:=imposs to fin do
     for j:=0 to 255 do
       table[i,chr(j)]:=imposs;
   for x:='a' to 'z' do
    begin
      table[0,x]:=1;  {  (q0,lettre) → q1  }
      table[1,x]:=1;  {  (q1,lettre) → q1  }
    end;
    for x:='0' to '9' do
      table[1,x]:=1;  {  (q1,chiffre) → q1  }
    table[1,finmot]:= EtatFinal;  {  (q1,#) → qf  }
end;
```

Programme utilisant la classe AutomateEF Reconnaissant le langage des identificateurs	
program ProjAutomEF**;** {$APPTYPE CONSOLE} **uses** SysUtils, UAutomEF **in** 'UAutomEF.pas'**;** **var** AEFD**:**AutomateEF**;** **begin** AEFD**:=** AutomateEF.Create(2)**;** AEFD.Mot**:=** 'v14bcd73' **;** AEFD.Analyser**;** readln**;** **end.**	*Exécution de ce programme sur l'identificateur* ***prix2*** *:* (0, p)--> 1 (1, r)--> 1 (1, i)--> 1 (1, x)--> 1 (1, 2)--> 1 (1, #)--> 2 chaine reconnue ! *Exécution de ce programme sur l'exemple* ***v14bcd73*** *:* (0, v)--> 1 (1, 1)--> 1 (1, 4)--> 1 (1, b)--> 1 (1, c)--> 1 (1, d)--> 1 (1, 7)--> 1 (1, 3)--> 1 (1, #)--> 2 chaine reconnue !

3.4 Exemple : les constantes numériques

On donne des diagrammes syntaxiques de construction des constantes décimales positives Pascal-like :

constante :



Construisons un programme Delphi reconnaissant de telles constantes en utilisant le procédé de génération manuelle.

Méthode de travail adoptée :*(identique à la précédente)*

- **Détermination d'une grammaire G adéquate.**
- **Construction de l'automate AEFD associé à G.**
- **Programme pascal associé à l'automate construit.**

Détermination d'une grammaire G_{cte} adéquate

Reconnaissons d'abord qu'il y a une grammaire de type 3 engendrant ces identificateurs.

Soient les ensembles :

EnsChiffre = { 0, 1,..., 9 }. *// les 10 chiffres*

V_T = EnsChiffre

V_N = { ⟨*constante*⟩, A, B }

Posons G_{cte} = (V_T , V_N , ⟨ *constante* ⟩ , Règles) une grammaire où

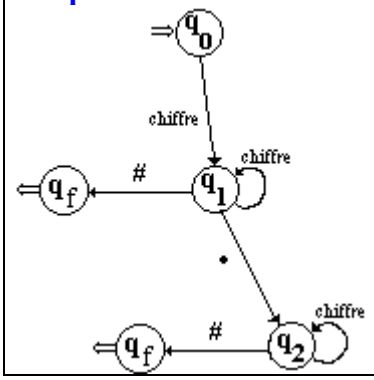
Axiome : ⟨*constante*⟩
Règles :
⟨*constante*⟩ → **0** |**1**|...|**9** A
A → **0** |**1**|...|**9** A
A → ε
A → B
B → **0** |**1**|...|**9** B
B → ε
La grammaire G_{cte} est de type 3 déterministe.

Construction de l'automate associé à G_{cte}

Afin de réduire le nombre de lignes de texte, nous adoptons les mêmes conventions d'écriture que celles de l'exemple précédent:

(q_k , Chiffre) → q_i , *représente l'ensemble des 10 règles :*
(q_k ,**0**) → q_i
.....
(q_k ,**9**) → q_i

Etats associés aux éléments de V_N:	**Règles de l'automate associé à G_{cte} :**
⟨*constante*⟩ ⇔ q_0 ⟨ A ⟩ ⇔ q_1 ⟨ B ⟩ ⇔ q_2 Etat initial = q_0 Etat final = q_f **Vocabulaire terminal de l'automate :** $V_T' = V_T \cup \{\#\}$ = Chiffre ∪ {#}	(q_0 , Chiffre) → q_1 *// 10 règles* (q_1 , Chiffre) → q_1 *// 10 règles* (q_1 ,#) → q_f (q_1 , .) → q_2 (q_2 , Chiffre) → q_2 *// 10 règles* (q_2 ,#) → q_f Soit un total de 33 règles.

Graphe de l'automate :	**Matrice de transitions de l'automate:**
	$\begin{pmatrix} & 0 & \cdots & 9 & . & \# \\ q_0 & q_1 & \cdots & q_1 & \blacksquare & \blacksquare \\ q_1 & q_1 & \cdots & q_1 & q_2 & q_f \\ q_2 & q_2 & \cdots & q_2 & \blacksquare & q_f \end{pmatrix}$

Programme associé à l'automate

Nous héritons de la classe AutomateAbstr. Comme dans l'exercice précédent , nous n'avons que la méthode **init_table** à redéfinir

```
AutomateEF = class ( AutomateAbstr )
 protected
   procedure init_table; override;
end;

implementation

procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  x:char;
 begin
  for i:=imposs to fin do
   for j:=0 to 255 do
     table[i,chr(j)]:=imposs;
```

```
  for x:='0' to '9' do
   begin
     table[0,x]:=1;  { (q0,chiffre) → q1  }
     table[1,x]:=1;  { (q1,chiffre) → q1  }
     table[2,x]:=2;  { (q2,chiffre) → q2  }
   end;
   table[1,'.']:=2;  { (q1,'.') → q2  }
   table[1,finmot]:= EtatFinal;  { (q1,#) →  qf  }
   table[2,finmot]:= EtatFinal;  { (q2,#) →  qf  }
end;
```

Programme utilisant la classe AutomateEF

Reconnaissant le langage des constantes numériques

```
program ProjAutomEF;

 {$APPTYPE CONSOLE}

 uses
  SysUtils,
  UAutomEF in 'UAutomEF.pas';
 var AEFD:AutomateEF;

  begin
   AEFD:= AutomateEF.Create(3);
   AEFD.Mot:= 145.78 ;
   AEFD.Analyser;
   readln;
  end.
```

Exécution de ce programme sur l'exemple ***145*** *:*

```
( 0, 1)-->  1
( 1, 4)-->  1
( 1, 5)-->  1
( 1, #)-->  3
chaine reconnue !
```

Exécution de ce programme sur l'exemple ***145.78*** *:*

```
( 0, 1)-->  1
( 1, 4)-->  1
( 1, 5)-->  1
( 1, .)-->  2
( 2, 7)-->  2
( 2, 8)-->  2
( 2, #)-->  3
chaine reconnue !
```

Le lecteur aura pu se convaincre de la facilité d'utilisation d'un tel générateur manuel. Il lui est conseillé de réécrire un programme personnel fondé sur ces idées. Le polymorphisme dynamique de méthode a montré son utilité dans ces exemples.

Nous appliquons dans le chapitre suivant cette connaissance toute neuve à un petit projet de construction d'un **interpréteur** pour un langage analysable par grammaire de type 3. Nous verrons comment utiliser le graphe d'un automate dans le but de programmer les décisions d'interprétation. Là aussi, le lecteur pourra aisément adapter la méthodologie à d'autres exemples semblables construits sur des K-grammaires.

6.3 classe d'interpréteur d'un langage de type 3

Objectif : Construction et utilisation d'une classe d'interpréteur d'un langage sans constantes, généré par une grammaire de type 3.

ENONCE :

On donne la Grammaire **G** suivante :

V_N = { ⟨prog.⟩, ⟨bloc⟩, ⟨égal⟩, ⟨suite1⟩, ⟨suite2⟩, ⟨membre droit⟩, ⟨plus⟩, ⟨suite3⟩ , ⟨moins⟩ , ⟨mult⟩ , ⟨div⟩ }
V_T = { **'a'** , **'b'**,..., **'z'** , **'}'** , **'{'** , **';'** , **'L'**, **'E'** , **'+'** , **'='**, **'*'** , **'-'** , **'/'** }

Axiome : ⟨prog.⟩
Règles :
⟨prog.⟩ → **{** ⟨bloc⟩
⟨bloc⟩ → **}**
⟨bloc⟩ → **a**⟨égal⟩ | **b**⟨égal⟩ | . . . | **z**⟨égal⟩
⟨bloc⟩ → **L**⟨suite1⟩ | **E**⟨suite1⟩
⟨suite1⟩ → **a**⟨suite2⟩ | **b**⟨suite2⟩ | . . . | **z**⟨suite2⟩
⟨suite2⟩ → **;** ⟨bloc⟩
⟨égal⟩ → **=** ⟨membre droit⟩
⟨membre droit⟩ → **a**⟨moins⟩ | **b**⟨moins⟩ | . . . | **z**⟨moins⟩
⟨membre droit⟩ → **a**⟨plus⟩ | **b**⟨plus⟩ | . . . | **z**⟨plus⟩
⟨membre droit⟩ → **a**⟨mult⟩ | **b**⟨mult⟩ | . . . | **z**⟨mult⟩
⟨membre droit⟩ → **a**⟨div⟩ | **b**⟨div⟩ | . . . | **z**⟨div⟩
⟨membre droit⟩ → **a**⟨reste⟩ | **b**⟨reste⟩ | . . . | **z**⟨reste⟩
⟨moins⟩ → **-** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨plus⟩ → **+** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨mult⟩ → ***** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨div⟩ → **/**⟨suite3⟩ | **;** ⟨bloc⟩
⟨reste⟩ → **%** ⟨suite3⟩ | **;** ⟨bloc⟩
⟨suite3⟩ → **a**⟨suite2⟩ | **b**⟨suite2⟩ | . . . | **z**⟨suite2⟩

Questions :

1°) Construire une classe interpréteur de L(G). On donne la sémantique suivante :
(les spécifications sont fournies en langage algorithmique)

Lx correspond à **lire**(x)
Ex correspond à **ecrire**(x)
x = y correspond à **x** ¬ y
a,b,...,z correspondent à des variables contenant des entiers relatifs
+ correspond à l'opérateur d'addition sur les entiers relatifs.
- correspond à l'opérateur de soustraction sur les entiers relatifs.
* correspond à l'opérateur de multiplication sur les entiers relatifs.
/ correspond à l'opérateur de quotient euclidien sur les entiers relatifs.
% correspond à l'opérateur de reste euclidien sur les entiers relatifs.

On utilisera une *mémoire centrale* dans laquelle se trouveront les variables (dans un tableau) et une autre table contenant les noms des variables et leur *adresse* en mémoire centrale (*table des symboles*).

2°) Construire une IHM de calculatrice programmable fondée sur la classe précédente d'interpréteur du langage L(G). calculatrice dans laquelle l'utilisateur peut entrer des lignes de programmes en L(G) et les exécuter.

UNE SOLUTION AU PROBLEME PROPOSE

Spécifications de base du logiciel

1°) Construction d'un analyseur **du langage L(G).**
2°) Construction d'un interpréteur **du langage L(G).**
3°) Le programme console d'interpréteur.
4°) Construction de 2 classes**.**

Démarche adoptée :
Nous adoptons la méthodologie de développement de l'algorithme d'interprétation évoquée au chapitre précédent (construction d'un analyseur puis de l'interpréteur associé), en l'adaptant au langage L(G). Ensuite nous construirons une classe fondée sur cet algorithme.

1°) Construction d'un analyseur du langage L(G)

Nous remarquons que la grammaire **G** est une grammaire de type 3 déterministe(d'état fini). En appliquant le principe de correspondance entre grammaires de type 3 et automates d'états finis, nous construisons l'AEFD associé qui sera un analyseur du langage engendré par **G**.

correspondance entre les éléments de $\mathbf{V_N}$ et les états de l'automate :
<prog.> ⇔ q_0
<suite2> ⇔ q_3
<plus> ⇔ q_6 *idem pour* <moins>, <mult>, et <reste>
<bloc> ⇔ q_1
<egal> ⇔ q_4
<suite3> ⇔ q_7
<suite1> ⇔ q_2
<membre droit> ⇔ q_5

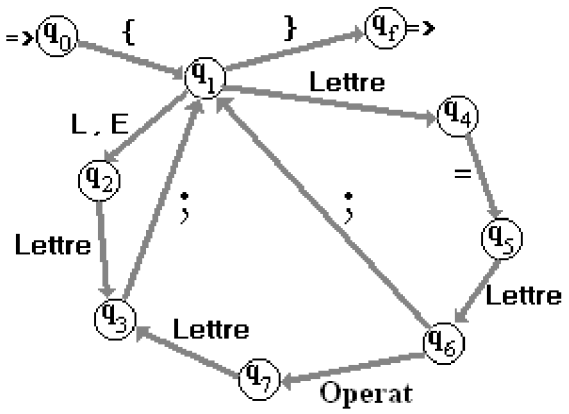
Soit l'AEFD A, $A = (V_T', E, q_0, q_f, \mu)$
$E = \{ \mathbf{q_0}, q_1, q_2, q_3, q_4, q_5, q_6, q_7, \mathbf{q_f} \}$
état initial : $\mathbf{q_0}$
état final : $\mathbf{q_f}$
V_T = { **'a'** , **'b'**,..., **'z'** , **'}'** , **'{'** , **';'** , **'L'**, **'E'** , **'+'** , **'='**, **'*'** , **'-'** , **'/'** }

Nous poserons pour simplifier :
Lettre = {'a', 'b',..., 'z'},
Operat = { '+' , '*' , '-' , '/' , '%' },

Afin de réduire le nombre de lignes de texte nous adoptons la convention d'écriture suivante :

(q_k , **Lettre**) → q_i représente l'ensemble des 26 règles	(q_k ,**a**) → q_i (q_k ,**b**) → q_i ... (q_k ,**z**) → q_i
(q_k , **Operat**) → q_i représente l'ensemble des 5 règles	(q_k ,+) → q_i (q_k , -) → q_i (q_k ,*) → q_i (q_k , /) → q_i (q_k ,%) → q_i

Nous obtenons alors un AEFD dont nous donnons les règles et le graphe.

Règles de transitions de µ	Graphe de l'automate A
(q_0, {) → q_1 (q_1, }) → q_f (q_1, **Lettre**) → q_4 (q_1, **L**) → q_2 (q_1, **E**) → q_2 (q_2, **Lettre**) → q_3 (q_3, **;**) → q_1 (q_4, **=**) → q_5 (q_5, **Lettre**) → q_6 (q_6, **;**) → q_1 (q_6, **Operat**) → q_7 (q_7, **Lettre**) → q_3	

Employons la démarche conseillée dans le cours pour construire la matrice de transition de l'analyseur AEFD, grâce à la méthode init_table de la classe implantant l'automate :

```
const
 imposs=-1;  fin=20;  finmot='#';
type
   T_etat=imposs..fin;
   Vt=char;
   T_transition=array[T_etat,char] of T_etat;

AutomateEF = class ( AutomateAbstr )
 protected
    procedure init_table; override;
end;

implementation

procedure AutomateEF.init_table;
 {initialisation de la table des transitions}
 var  i:T_etat;  j:0..255;  k:char;
 begin
   for i:=imposs to fin do
     for j:=0 to 255 do
        table[i,chr(j)]:=imposs;

   table[0,'{']:=1;
   table[1,'}']:=fin;
   for k:='a' to 'z' do
   begin
      table[1,k]:=4;
      table[2,k]:=3;
      table[5,k]:=6;
      table[7,k]:=3;
   end;
   table[1,'E']:=2;
   table[1,'L']:=2;
   table[3,';']:=1;
   table[4,'=']:=5;
   table[6,'+']:=7;
   table[6,'*']:=7;
   table[6,'-']:=7;
   table[6,'%']:=7;
   table[6,'/']:=7;
   table[6,';']:=1;
end;
```

Programme d'analyseur utilisant la classe AutomateEF

Reconnaissant le langage L(**G**)

```
program ProjAutomEF;

 {$APPTYPE CONSOLE}

 uses
  SysUtils,
  UAutomEF in 'UAutomEF.pas';
 var AEFD:AutomateEF;

  begin
    AEFD:= AutomateEF.Create(8);
    AEFD.Mot:= '{La;Lb;Ea;a=b*c;}';
    AEFD.Analyser;
    readln;
  end.
```

Exécution de ce programme sur le mot: **{La;Lb;Ea;a=b*c;}**

```
( 0, {)--> 1
( 1, L)--> 2
( 2, a)--> 3
( 3, ;)--> 1
( 1, L)--> 2
( 2, b)--> 3
( 3, ;)--> 1
( 1, E)--> 2
( 2, a)--> 3
( 3, ;)--> 1
( 1, a)--> 4
( 4, =)--> 5
( 5, b)--> 6
( 6, *)--> 7
( 7, c)--> 3
( 3, ;)--> 1
( 1, })--> 8
chaine reconnue !
```

Exécution de ce programme sur le mot: **{La;Lb;Ea=b;}**

```
( 0, {)--> 1
( 1, L)--> 2
( 2, a)--> 3
( 3, ;)--> 1
( 1, L)--> 2
( 2, b)--> 3
( 3, ;)--> 1
( 1, E)--> 2
( 2, a)--> 3
( 3, =)--> -1
blocage, chaine non reconnue !
```

2°) Construction d'un interpréteur à partir de l'AEFD

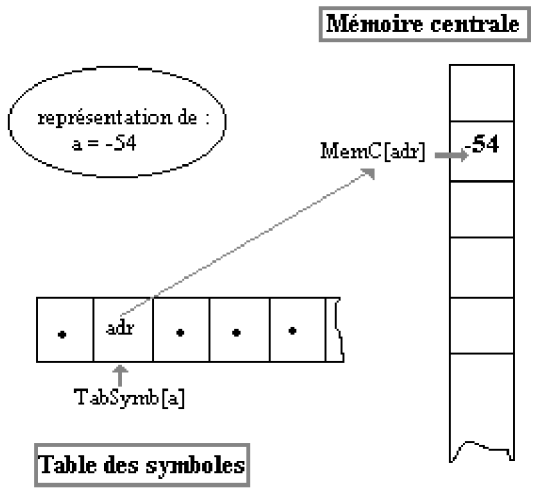
Rappelons les spécifications proposées par l'énoncé :

Lx correspond à **lire**(x)
Ex correspond à **ecrire**(x)

x = y correspond à **x** ← y
a,**b**,..., **z** correspondent à des variables entiers relatifs

+ correspond à l'opérateur d'addition sur les entiers relatifs.
- correspond à l'opérateur de soustraction sur les entiers relatifs.
* correspond à l'opérateur de multiplication sur les entiers relatifs.
/ correspond à l'opérateur de quotient euclidien sur les entiers relatifs.
% correspond à l'opérateur de reste euclidien sur les entiers relatifs.

Le mécanisme d'accès est simple



Spécifications opérationnelles

L'énoncé nous propose une spécification d'implantation en Delphi pour la mémoire centrale (nous la dénommons **MemC**)et la table des symboles (nous la dénommons **TabSymb**).

Le tableau **TabSymb** est indexé directement sur les caractères (ce qui évite la construction d'une fonction de codage). Chaque cellule **TabSymb**['a'], **TabSymb**['b'],..., **TabSymb**['z'], contient un nombre qui est l'adresse adr (numéro de case dans **MemC**) de la variable a, b,..,z.

A partir de ce numéro de case adr, la cellule MemC[adr] du tableau MemC contient la valeur numérique de la variable d'adresse adr.

Implantation Delphi de ces spécifications de données

```
const
 adresse=0..maxadresse;
type
 Symbole=char;
 T_mem=array[adresse] of integer;
 T_symb=array[Symbole]of adresse;
var
 MemC:T_mem; // la mémoire centrale
 TabSymb:T_symb; // la table des symboles
```

Spécifications d'implantation pour les instructions

En partant des spécifications de données précédentes, nous pouvons proposer une implantation de chacune des instruction du langage L(G).

Nous noterons par la suite, ≅ la relation de traduction en pseudo Delphi.

Nous avons utilisé trois métasymboles x,y et z pour remplacer le nom d'une quelconque des variables **a**, **b**…etc. afin de ne pas alourdir la traduction par de nombreuses lignes redondantes.

Interprétation de l'instruction L :

Lx	≅	TabSymb[x] := adressecourante ; adressecourante := adressecourante +1 ; readln(MemC[TabSymb[x]])

Interprétation de l'instruction E :

Ex	≅	adressecourante := TabSymb[x]; writeln(MemC[adressecourante])

Interprétation de l'instruction x = y :

x = y	≅	MemC[TabSymb[x]]:= MemC[TabSymb[y]]

*Interprétation de l'instruction x = y **oper** z :*

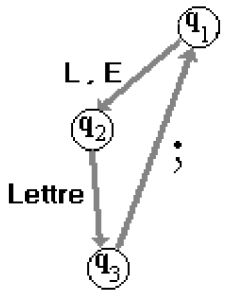
`x=y oper z` ≅ ***oper** est l'un des opérateurs suivants :* *{ '+', '*', '-', '/' }*	MemC[TabSymb[x]]:= MemC[TabSymb[y]] **oper** MemC[TabSymb[z]]

Nous allons construire notre interpréteur à partir du graphe de l'AEFD que nous possédons.

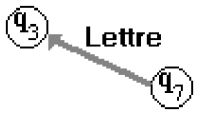
Ainsi, nous passerons de la version analyseur à la version interpréteur en suivant les chemins du graphe et en repérant les points clefs où l'interprétation d'une instruction est possible. Nous adoptons comme stratégie le fait que le lancement d'une interprétation ne sera possible que lorsque l'on sera arrivé dans le graphe à un endroit où une instruction vient juste d'être reconnue.

Interprétation des instructions Ex et Lx

Nous observons tout d'abord sur la portion de graphe de l'AEFD comment les instructions Ex et Lx sont reconnues.



Une telle instruction est entièrement reconnue lorsque l'automate est à l'état q_3. On pourrait lancer l'interprétation de Ex ou Lx, mais ce serait une erreur car il y a un autre chemin dans le graphe qui amène à l'état q_3.



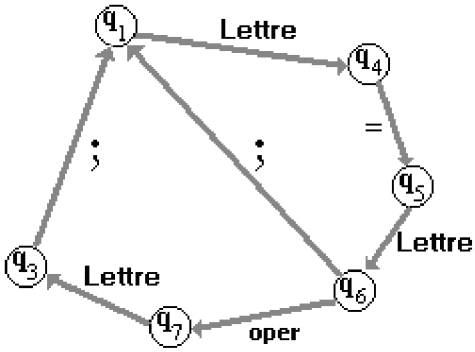
Nous voyons qu'il n'y a que deux manières différentes d'arriver en q_3 : soit en venant de q_2, soit en venant de q_7.

> Il nous faut une variable que nous nommerons **etatavant** qui mémorisera l'état précédent. Le symbole qui vient d'être analysé est stocké dans une variable que nous nommons **symlu**. D'autre part, lorsque nous sommes en q_3, le symbole qui vient d'être analysé est un élément de {a,b,c}. Afin de décider s'il s'agit d'une instruction Ex ou bien Lx, il nous faut avoir mémorisé le symbole précédent qui a été analysé en passant de q_1 à q_2. Nous nommerons cette variable **symprec**.

Voici donc en pseudo-Delphi le code de lancement de l'interprétation de Lx ou de Ex.
 	
Ce code est à rajouter à l'AEFD précédent.
	If (etat=q_3)and(etatavant=q_2)**and**(symprec=L)**then**
	begin *{on interprète le Lx}*
	TabSymb[symlu] := adressecourante ;
	adressecourante := adressecourante +1 ;
	readln(MemC[TabSymb[symlu]])
	end
	else *{on interprète le Ex}*
	begin
	adressecourante := TabSymb[symlu];
	writeln(MemC[adressecourante])
	end

Interprétation de l'instruction x = y oper z

Nous avons vu que l'autre façon d'arriver à l'état q_3 est d'avoir suivi l'arc q_7 à q_3.



Ce chemin correspond très exactement à l'analyse d'une instruction x = y **oper** z. Afin de pouvoir interpréter correctement cette instruction, nous devons avoir mémorisé les noms des variables x, y et z le long du chemin d'analyse : $q_1 \rightarrow q_4 \rightarrow q_5 \rightarrow q_6 \rightarrow q_7 \rightarrow q_3$. Afin de ne pas rajouter trop de nouveaux états nous avons décidé de n'avoir qu'une transition sur un ensemble d'opérateurs (q_6 **oper**) → q_7. Nous regroupons alors les symboles +,-,*,/,% dans un même ensemble (**Operat** : set of Vt), que nous initialiserons dans la procédure d'initialisation :
Operat =['+', '-', '*', '/', '%' **]**

Nous notons que :
x est connu lorsque l'AEFD est à l'état q_4 (à la fin de l'arc $q_1 \rightarrow q_4$)
y est connu lorsque l'AEFD est à l'état q_6 (à la fin de l'arc $q_5 \rightarrow q_6$)
z est connu lorsque l'AEFD est à l'état q_3 (à la fin de l'arc $q_7 \rightarrow q_3$)

Le stockage d'un troisième symbole de variable nécessite l'utilisation d'une nouvelle variable servant à le mémoriser. Nous la nommerons **symavant**.

En résumant la situation, nous aurons pour x = y **oper** z :

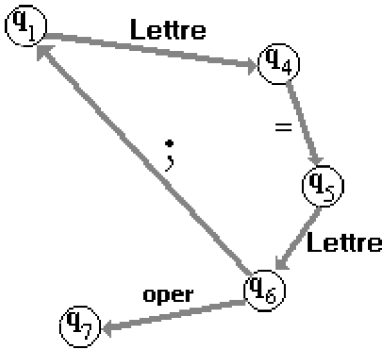
- z est stocké dans **symlu**, il est reconnu à l'état q_3.
- y est stocké dans **symprec**, il est reconnu à l'état q_6.
- x est stocké dans **symavant**, il est reconnu à l'état q_4.
- l'opérateur **oper** est reconnu à l'état q_7,il est alors rangé dans une variable **OperVar** de type Vt servant à cet effet.

Voici donc en pseudo-Delphi le code de lancement de l'interprétation de l'instruction x = y **oper** z. Ce code est aussi à rajouter au code précédent.

```
If etat=q4 then symavant := symlu ;
If etat=q6 then symprec := symlu ;
If etat=q7 then
  If symlu in Operat then   OperVar := symlu;
If (etat=q3)and(etatavant=q7)then {on interprète x = y oper z}
case OperVar of
 '+':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]+MemC[TabSymb[symlu]];
 '-':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]-MemC[TabSymb[symlu]];
 '*':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]*MemC[TabSymb[symlu]];
 '/':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]div MemC[TabSymb[symlu]];
 '%':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]mod MemC[TabSymb[symlu]];
end;
```

Interprétation de l'instruction x=y

Si nous observons la partie du graphe de l'AEFD correspondant à l'analyse de l'instruction x=y, nous remarquons que contrairement aux cas précédents ce n'est pas à l'état q_6 que nous pouvons prendre une décision.



En effet, à partir de q_6 il y a deux possibilités d'instructions : soit x=y, soit x = y **oper** z. Il nous faut aller un état plus loin dans le graphe (déterministe) afin de décider si l'on est dans un cas ou dans l'autre.

Si l'état d'après q_6 est q_1, il s'agit d'une instruction x=y, si l'état d'après q_6 est q_7 il s'agit d'une instruction x=y+z.

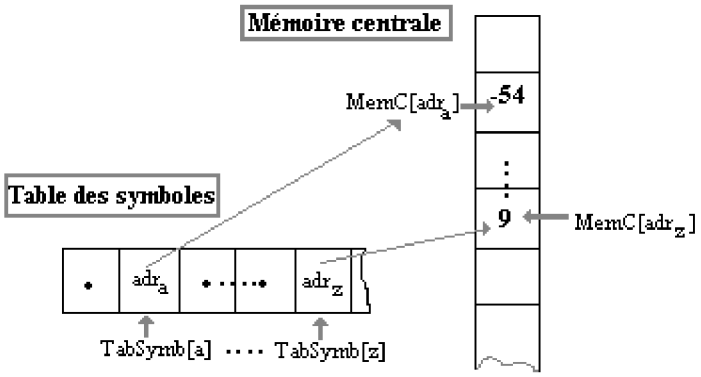
Le chemin d'analyse d'une instruction x=y est :
$q_1 \rightarrow q_4 \rightarrow q_5 \rightarrow q_6 \rightarrow q_1$.
Comme dans l'analyse du problème précédent nous avons :

- **;** est stocké dans **symlu**, il est reconnu à l'état q_1.
- y est stocké dans **symprec**, il est reconnu à l'état q_6.
- x est stocké dans **symavant**, il est reconnu à l'état q_4.

Voici donc en pseudo-Delphi, le code de lancement de l'interprétation de l'instruction x=y. Ce code est le dernier à rajouter au code précédent.

```
If (etat=q1)and(etatavant=q6)then {on interprète x = y}
   MemC[TabSymb[symavant]] := MemC[TabSymb[symprec]]
```

Le mécanisme d'accès implanté
Par TabSymb et MemC :



Le lecteur pourra étendre le programme en augmentant par exemple le nombre de variables, ou le nombre d'opérateurs.

Voici ci-dessous les squelettes des méthodes Delphi d'une classe d'interpréteur **InterpreteurLang** permettant d'étendre l'analyseur en interpréteur :

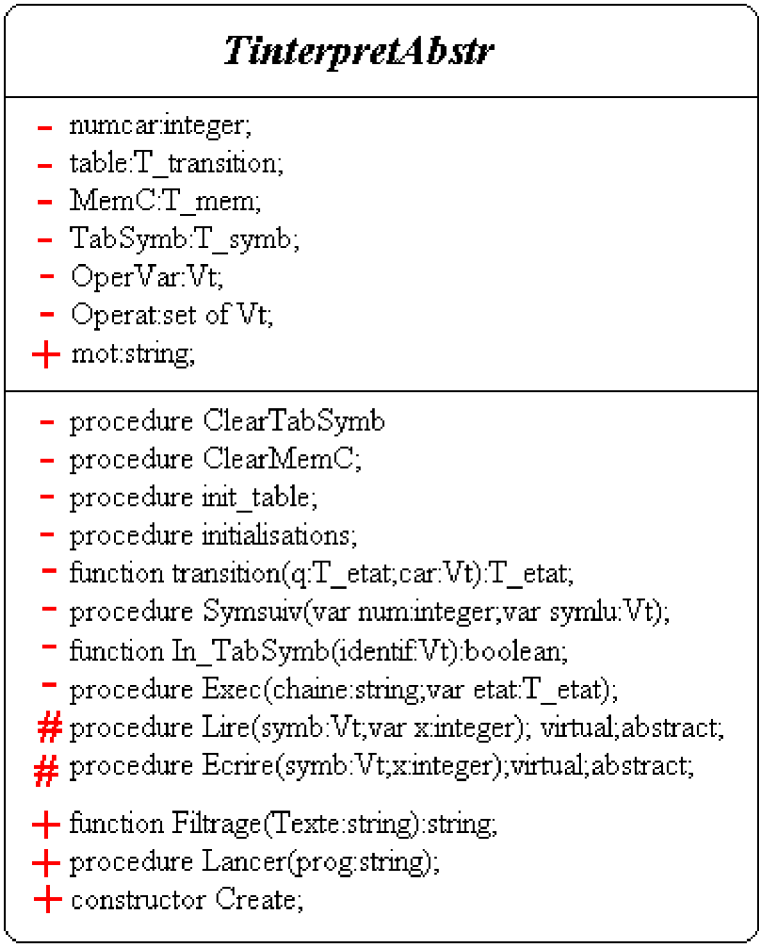
```
type
  T_etat=imposs..fin;    Vt=char;

procedure InterpreteurLang.init_table;
{initialisation de la table des transitions de l'automate AEFD }

procedure InterpreteurLang.ClearMemC;
 // RAZ mémoire centrale

procedure InterpreteurLang.ClearTabSymb;
 // RAZ table des symboles

 procedure InterpreteurLang.initialisations;
 // RAZ tout

{-------------------- INTERPRETEUR -------------------}

procedure InterpreteurLang.Interprete(chaine:string;var etat:T_etat);
{moteur d'analyse et d'interpretation de l'automate}

 function InterpreteurLang.transition(q:T_etat;car:Vt):T_etat;
 {par la table de transition :}

procedure InterpreteurLang.Symsuiv(var num:integer;var symlu:Vt);
{fournit le symbole suivant à analyser}

function InterpreteurLang.In_TabSymb(identif:Vt):boolean;
{indique si le symbole identif est deja dans TabSymb}
```

Comme notre interpréteur doit lire et analyser un texte de programme et écrire les resultats d'exécution, nous proposons d'écrire une classe qui contienne toutes spécifications nécessaires et utiles au fonctionnement d'un interpréteur, mais qui ne dépende pas de la façon dont on lit et dont on affiche les résultats. Pour cela nous construisons une classe abstraite **TinterpretAbstr** qui possède 2 méthodes abstraites (donc non implémentées) **Lire** et **Ecrire**. Le soin d'expliquer comment lire ou écrire est délégué à une classe **'concrète'** qui dérivera de **TinterpretAbstr**.

1°) Construction d'une classe abstraite d'interpréteur de L(G)

Nous reprenons toutes les procédures et fonctions du programme console et nous les transformons en méthodes de classe. Ci-dessous un diagramme UML-like de la signature de la classe **TinterpretAbstr**. Nous ajoutons à cette classe, en visibilité **protected,** les deux méthodes abstraites :

```
procedure Lire(symb:Vt;var x:integer); // symb = la variable à lire, x = sa valeur entière
procedure Ecrire(symb:Vt; x:integer);  // symb = la variable à ecrire, x = sa valeur entière
```



Nous reprenons dans une unit Delphi que nous nommons **Uinterpreteur,** les mêmes déclarations de constantes et de type que dans le programme console :

```
Unit Uinterpreteur :
    interface
    const
      imposs=-1;
      fin=8;
      finmot='#';
      maxadresse=100;
    type
     T_etat=imposs..fin;
     Vt=char;
     T_transition=array[T_etat,char] of T_etat;
     adresse=0..maxadresse;
     Symbole=char;
     T_mem=array[adresse] of integer;
     T_symb=array[Symbole]of adresse;
```

Ce qui nous donne dans **Uinterpreteur** la signature suivante pour la classe abstraite TinterpretAbstr :

```
    TinterpretAbstr=class
      private
       numcar:integer;
       table:T_transition;
       MemC:T_mem;
       TabSymb:T_symb;
```

```
    OperVar:Vt;
    Operat : set of Vt;
    EtatFinal : T_etat;
    procedure ClearTabSymb;
    procedure ClearMemC;
    procedure init_table;
    procedure initialisations;
    function transition(q:T_etat;car:Vt):T_etat;
    procedure Symsuiv(var num:integer;var symlu:Vt);
    function In_TabSymb(identif:Vt):boolean;
    procedure Exec(chaine:string;var etat:T_etat);
  protected
    procedure Lire(symb:Vt;var x:integer); virtual;abstract;
    procedure Ecrire(symb:Vt;x:integer); virtual;abstract;
  public
    mot:string;
    function Filtrage(Texte:string):string;
    procedure Lancer(prog:string);
    constructor Create(fin : T_etat) ;
  end;
```

Nous avons rajouté une méthode de filtrage du texte source permettant une saisie plus libre du texte (avec des blancs, des sauts de ligne,...) et épurant le texte entré de tous ces éléments :

function Filtrage(Texte: **string**): **string**;

Le texte suivant entré au clavier sur 8 lignes, soumis à la méthode de filtrage est restitué pour analyse une fois épuré :

{
Lb; Lc;
a = c + b ;
d=a*c;
Ea; Eb;
Ec;
Ed;
}#
	{Lb;Lc;a=c+b;d=a*c;Ea;Eb;Ec;Ed;}#

Nous donnons ici la partie **implementation** de la unit **Uinterpreteur** avec les corps des méthodes :

```
{ TinterpretAbstr }

procedure TinterpretAbstr.ClearMemC;
// RAZ mémoire centrale
var i:adresse;
begin
 for i:=0 to maxadresse do
  MemC[i]:=0;
end;
```

```
procedure TinterpretAbstr.ClearTabSymb;
// RAZ table des symboles
var i:Symbole;
begin
 for i:=chr(0) to chr(255) do
  TabSymb[i]:=0;  {0 indique : variable non definie}
end;

procedure TinterpretAbstr.init_table;
{initialisation de la table des transitions de l'automate AEFD }
var
  i:T_etat;   j:0..255;   k:char;
begin
 for i:=imposs to fin do
   for j:=0 to 255 do
   table[i,chr(j)]:=imposs;{par défaut tout est non reconnu}
 table[0,'{']:=1;
 table[1,'}']:= EtatFinal;
 for k:='a' to 'z' do
 begin
  table[1,k]:=4;
  table[2,k]:=3;
      table[5,k]:=6;
  table[7,k]:=3;
 end;
 table[1,'E']:=2;
 table[1,'L']:=2;
 table[3,';']:=1;
 table[4,'=']:=5;
 table[6,'+']:=7;
  table[6,'*']:=7;
 table[6,'-']:=7;
 table[6,'%']:=7;
 table[6,'/']:=7;
 table[6,';']:=1;
end;

procedure TinterpretAbstr.initialisations;
// RAZ tout
begin
  ClearMemC;
  ClearTabSymb;
  init_table;
  Operat :=['+', '-', '*', '/', '%' ]
end;

function TinterpretAbstr.transition(q: T_etat; car: Vt): T_etat;
{par la table de transition :}
begin
 result:=table[q,car];
end;

procedure TinterpretAbstr.Symsuiv(var num:integer;var symlu:Vt);
 {fournit le symbole suivant à analyser}
begin
  if num<=length(mot) then
  begin
   symlu:=mot[num];
   num:=num+1
```

(q0, '[') --> q1
(q1, ']') --> qf
(q1, Lettre) --> q4
(q1, L) --> q2
(q1, E) --> q2
(q2, Lettre) --> q3
(q3, ;) --> q1
(q4, =) --> q5
(q5, Lettre) --> q6
(q6, ;) --> q1
(q6, Operat) --> q7
(q7, Lettre) --> q3

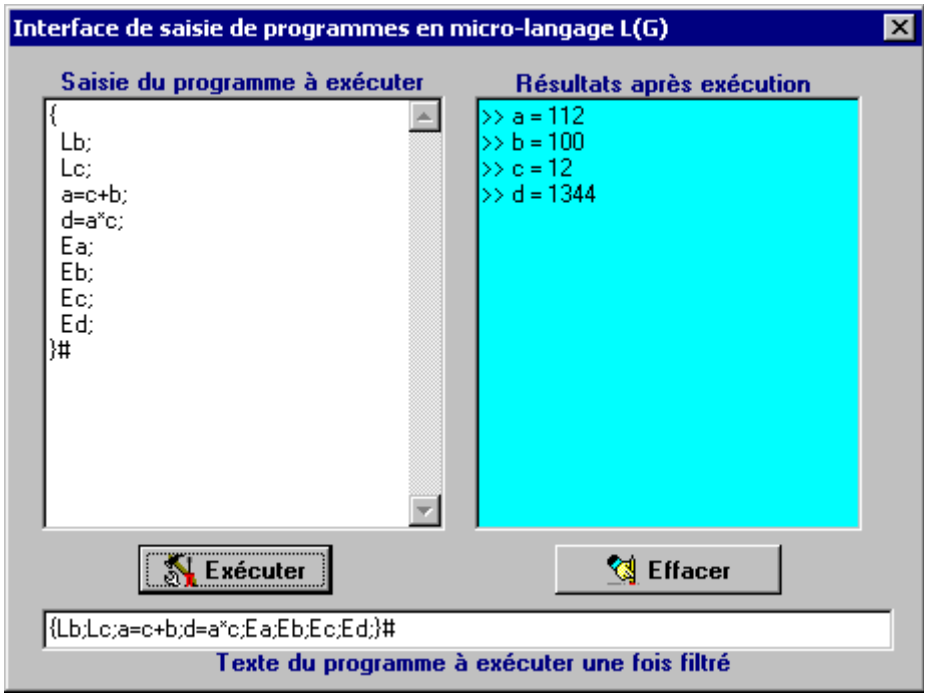
```
   end
   else          {si on veut lire apres la fin}
    symlu:=chr(0);{caractere NUL invisible     }
  end;

  function TinterpretAbstr.In_TabSymb(identif:Vt):boolean;
   {indique si le symbole identif est deja dans TabSymb}
  begin
   if TabSymb[identif]=0 then In_TabSymb:=false
   else In_TabSymb:=true
  end;

  procedure TinterpretAbstr.Exec(chaine: string; var etat: T_etat);
 {moteur d'analyse et d'interpretation de l'automate}
  var
    symlu:Vt;
    adressecourante:adresse;
    symprec,symavant:Vt;
    etavant:T_etat;
 begin {Interpreteur - Exec}
  numcar:=1;
  etat:=0;
  adressecourante:=1;  {on commence toujours a 1}
  mot:= chaine;
  while (etat<>imposs)and(etat<>fin) do
  begin
   Symsuiv(numcar,symlu);
   etavant:=etat;
   etat:=transition(etat,symlu);
   {------------ partie due a l'interpretation ------------}
   if (etat=2)then
     symprec:=symlu;
   if etat=4 then
   begin
     symavant:=symlu;
     if not In_TabSymb(symlu)then
     begin
      TabSymb[symlu]:=adressecourante;
      adressecourante:=adressecourante+1
     end
   end;
   if etat=6 then
     symprec:=symlu;
   if etat=7 then
    if symlu in Operat then
      OperVar := symlu;
   if (etat=3)and(etavant=2)and(symprec='L') then { Lx; }
   begin
     TabSymb[symlu]:=adressecourante;
     adressecourante:=adressecourante+1;
     Lire(symlu,MemC[TabSymb[symlu]]);
   end
   else
   if (etat=3)and(etavant=2)and(symprec='E') then { Ex; }
   begin
    adressecourante:=TabSymb[symlu];
    Ecrire(symlu,MemC[adressecourante]);
   end
   else
   if (etat=1)and(etavant=6)then              { x=y; }
```

```
    begin
     MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]
    end
    else
    if (etat=3)and(etavant=7)then              { x = y oper z; }
     case OperVar of
      '+':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]+MemC[TabSymb[symlu]];
      '-':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]-MemC[TabSymb[symlu]];
      '*':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]*MemC[TabSymb[symlu]];
      '/':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]div MemC[TabSymb[symlu]];
      '%':MemC[TabSymb[symavant]]:=MemC[TabSymb[symprec]]mod MemC[TabSymb[symlu]];
     end;
    {-------------------------------------------------------}
   end;
  end;{Interpreteur - Exec}

  function TinterpretAbstr.Filtrage(Texte: string): string;
  { filtrage du texte à interpréter :conservation des éléments
   du vocabulaire Vt et élimination des autres }
  var s:string;
    i:integer;
  begin
   s:='';
   for i:=1 to length(Texte) do
    if Texte[i] in ['a'..'z',';','{','}','L','E','=',finmot]+Operat  then
     s:=s+ Texte[i];
   result:=s
  end;


  procedure TinterpretAbstr.Lancer(prog: string);
  { uniquement pour appeler la méthode privée Exec
    et envoyer des messages à l'utilisateur selon
    la manière dont s'est passée l'exécution.
  }
  var q:T_etat;
  begin
   Exec(prog,q);
   if q=imposs then
     MessageDlg('Blocage, erreur de syntaxe !'+
     #13+#10+copy(prog,1,numcar)+'<<--', mtError ,[mbOk], 0)
   else
    if q=fin then
     MessageDlg('Exécution terminée !', mtInformation  ,[mbOk], 0);
  end;

 {----  le constructeur TinterpretAbstr ----}
 constructor TinterpretAbstr.Create(fin : T_etat) ;
 begin
    if fin in [1..20] then   EtatFinal:=fin
    else     EtatFinal:=20;
    initialisations;
 end;

end. {----  fin de la unit  ----}
```

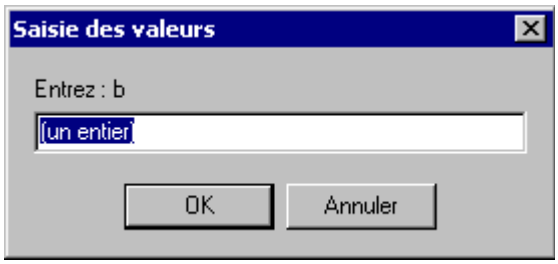
2°) Construction de deux classes héritant de TinterpretAbstr

Application IHM - première classe dérivée
Nous voulons écrire **une application IHM** utilisant la classe d'interpréteur que nous venons de construire, selon par exemple le modèle ci-dessous :

Nous afficherons les résultats par une **surcharge dynamiqu**e (redéfinition) de la méthode Ecrire à travers un TMemo :



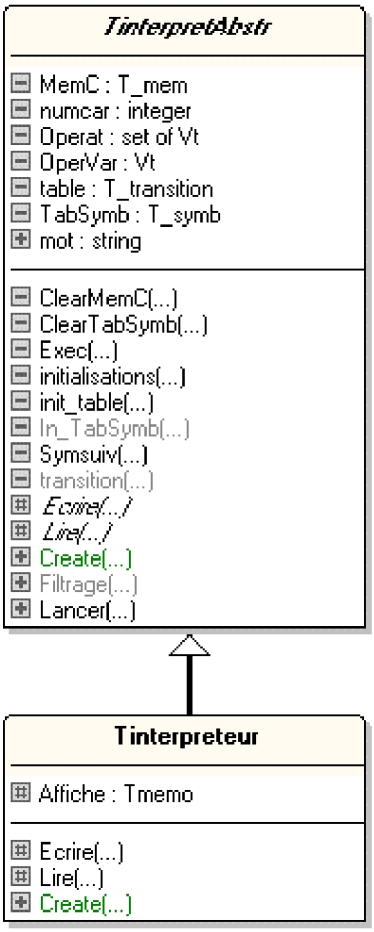
Nous saisirons les valeurs par une redéfinition de la méthode Lire à travers une inputBox :



Soit la classe Tinterpreteur héritant de notre classe abstraite, qui reçoit lors de la construction d'un objet d'interpréteur la référence d'un Tmemo déjà instancié dans l'IHM afin d''afficher les résultats :

```
{ Tinterpreteur cas de Delphi en mode application-IHM }
```

::UInterprete

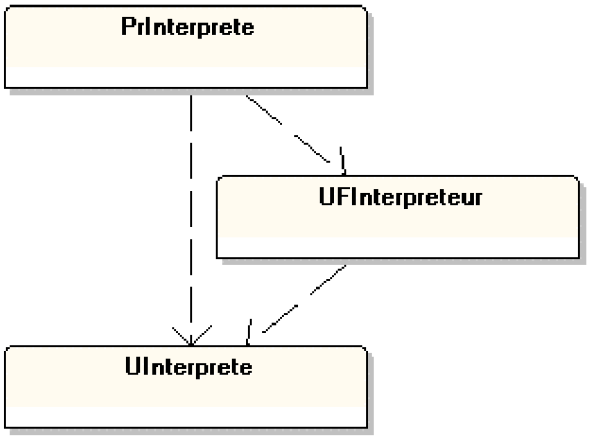


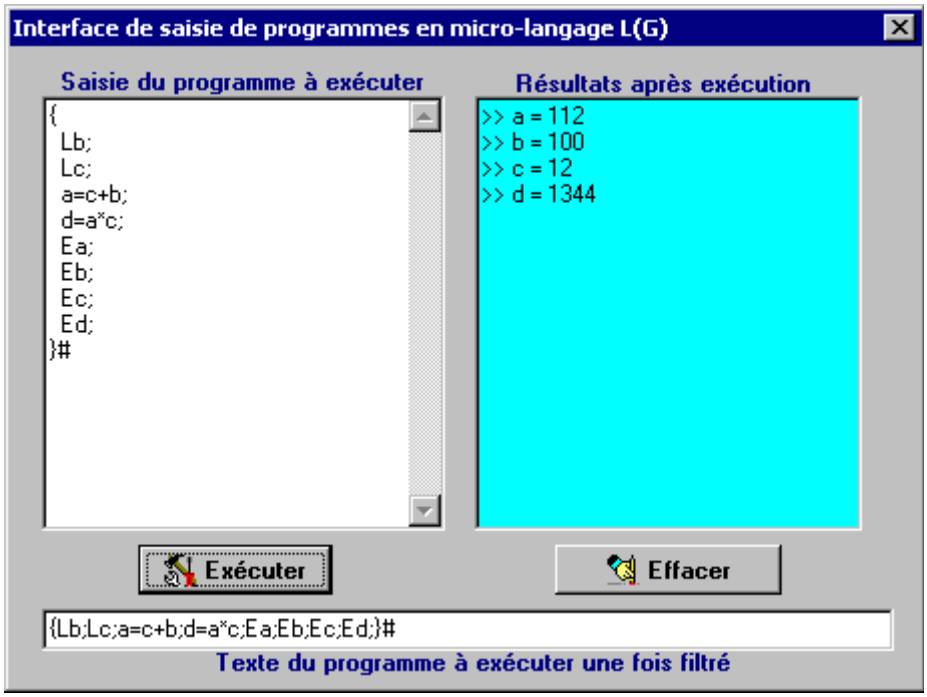

```
Tinterpreteur=class(TinterpretAbstr)
   protected
    Affiche:Tmemo;
    procedure Lire(symb:Vt;var x:integer); override;
    procedure Ecrire(symb:Vt;x:integer); override;
   public
    constructor Create(sortie:TMemo);
  end;
```

```
procedure Tinterpreteur.Ecrire(symb:Vt;x:integer);
begin
 Affiche.Lines.Append('>> '+symb+' = '+inttostr(x))
end;

procedure Tinterpreteur.Lire(symb:Vt;var x: integer);
var InputString:string;
begin
 InputString:= InputBox('Saisie des valeurs', 'Entrez : '+symb, '(un entier)');
 try
  x:=strtoint(InputString);
 except
  x:=1;
 end
```

```
end;

{---- le constructeur Tinterpreteur ----}
constructor Tinterpreteur.Create(sortie: TMemo);
begin
 inherited Create;
 Affiche:=sortie
end;
```

Terminons en livrant le code source et l'organisation de l'IHM de saisie et de traitement (exe et projet complet) :

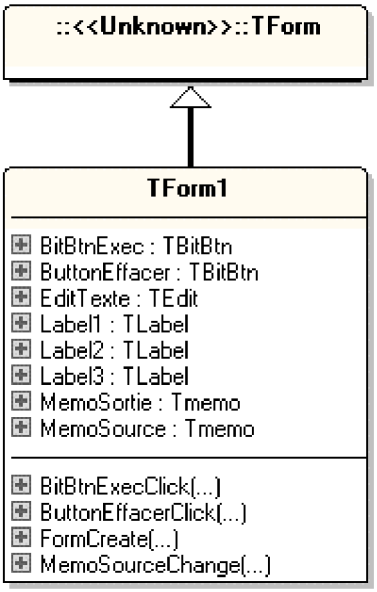





```
unit UFInterpreteur;

interface

uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls,UInterprete, Buttons;
```

::UFInterpreteur




```
type
  TForm1 = class(TForm)
    MemoSource: TMemo;
    MemoSortie: TMemo;
    EditTexte: TEdit;
    BitBtnExec: TBitBtn;
    ButtonEffacer: TBitBtn;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure MemoSourceChange(Sender: TObject);
    procedure BitBtnExecClick(Sender: TObject);
    procedure ButtonEffacerClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;
  interpreteur:Tinterpreteur; { objet interpréteur }

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
{ instanciation de l'objet interpréteur
  et liaison avec le Tmemo MemoSortie
}
begin
 interpreteur:=Tinterpreteur.Create(MemoSortie);
```

```
 EditTexte.Text:= interpreteur.Filtrage(MemoSource.text);
end;

procedure TForm1.MemoSourceChange(Sender: TObject);
{ filtrage du texte sur l'événement OnChange du TMemo
  et stockage dans le TEdit EditTexte }
begin
 EditTexte.Text:= interpreteur.Filtrage(MemoSource.text);
end;

procedure TForm1.BitBtnExecClick(Sender: TObject);
{ OnClick du TBitBtn BitBtnExec "Exécuter" }
begin
 interpreteur.Lancer(EditTexte.Text);
end;

procedure TForm1.ButtonEffacerClick(Sender: TObject);
{ OnClick du TBitBtn BitBtnEffacer "Effacer" }
begin
 MemoSortie.Clear
end;

end.
```

Application console - seconde classe dérivée
Nous voulons maintenant écrire **une application console** utilisant la classe d'interpréteur TinterpretAbstr, selon le modèle ci-dessous :

```
D:\_coursinfo34\ChapProjets\Interpreteur\Console\InterpreteurLang.exe
Interpreteur du micro-langage
Vt = a, b,...,z, }, =, +, /, *, -, %, L, E, {, ;
Entrez un programme entre deux{ },termine par un "#"
Exemples:
---------
prog-1: {La;Lb;b=c;Ea;Eb;}#
prog-2: {La;}#
prog-3: {a=b+c;}#
prog-4: {Lb;Lc;a=c+b;d=a*c;Ea;Eb;Ec;Ed;}#
****************************************
{Lb;Lc;a=c+b;d=a*c;Ea;Eb;Ec;Ed;}#
****** INTERPRETE micro-langage ********
entrez b :100
entrez c :12
>> a = 112
>> b = 100
>> c = 12
>> d = 1344
```

Nous afficherons les résultats par une redéfinition de la méthode Ecrire à travers la procédure **writeln.**

Nous saisirons les valeurs par une redéfinition de la méthode Lire à travers la procédure **readln.**

Soit la nouvelle classe Tinterpreteur héritant de notre classe abstraite :

```
{ Tinterpreteur cas de Delphi en mode console }

Tinterpreteur=class(TinterpretAbstr)
   protected
    procedure Lire(symb:Vt;var x:integer); override;
    procedure Ecrire(symb:Vt;x:integer); override;
  end;
```

```
procedure Tinterpreteur.Ecrire(symb:Vt;x:integer);
begin
 writeln('>> ',symb,' = ', x)
end;

procedure Tinterpreteur.Lire(symb:Vt;var x: integer);
begin
 write('Entrez : ',symb,' :');
 readln(x)
 try
  readln(x);
 except
  x:=1;
 end
end;
```

A titre d'exercice simple, il vous est demandé d'écrire une unit Uinterprete contenant la classe Tinterpreteur précédente, puis le programme principal en Delphi mode console utilisant cette classe :

```
program InterpreteurLang;


{$APPTYPE CONSOLE}
uses sysutils , Uinterprete ;
var
  interpreteur:Tinterpreteur; { objet interpréteur }
begin
 interpreteur:=Tinterpreteur.Create(8);
  ....
 readln(mot);
 interpreteur.Lancer(mot);
end.
```

6.4 Mini-projet : réalisation d'un indentateur de code

Objectif : Utiliser les compétences aquises sur les structures de données, les tris et la notion d'automate à pile de mémoire pour construire un éditeur d'indentation de code du langage Delphi lui-même. Nous essayerons de dégager dans cet exemple, des éléments de construction qui pourront s'appliquer à d'autres langages classiques.

Soit le texte Delphi suivant (une implémentation de méthode) saisie par une personne :

```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin
if x < 5 then
y := y-7
else
while x in [a..b] do
begin
if expression(x,y)<'x' then
for x:=1 to 15 do
if expression(x,y)<'x' then
if expression(x,y)<'x' then
begin
x:=4;
end
else
if expression(x,y)<'x' then
begin
x:=4;
end
if expression(x,y)<'x' then
a:="bonjour"      //ceci est un commentaire
else   //ceci est un autre commentaire
wwww;
while dffg hh do
if dhdhdh then
dfdf
else
begin
b:="aaaaa"+"bbbbbb"+'c';
end
end;
end
end.
```

Soit le même texte saisie par une autre personne :

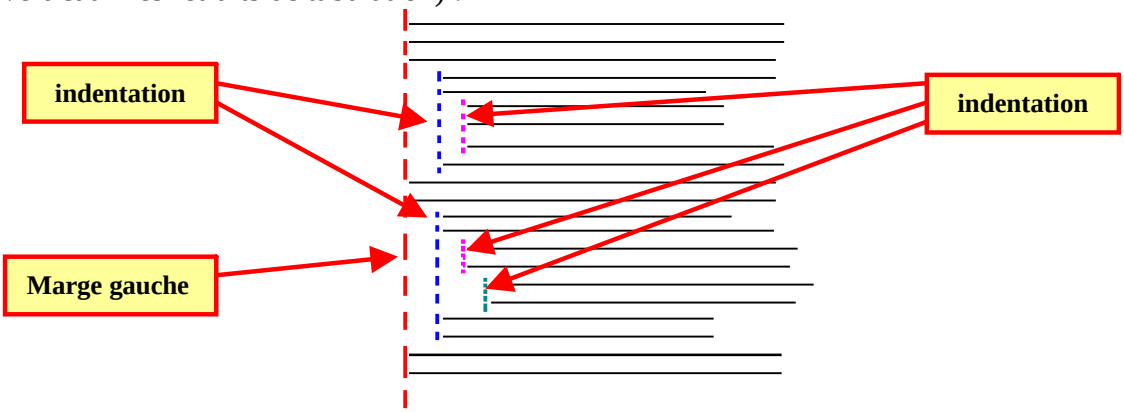
```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin if x < 5 then y := y-7 else
        while x in [a..b] do
    begin if expression(x,y)<'x' then for x:=1 to 15 do
  if expression(x,y)<'x' then
            if expression(x,y)<'x' then begin x:=4;
end else    if expression(x,y)<'x' then begin x:=4; end
    if expression(x,y)<'x' then  a:="bonjour"      //ceci est un commentaire
                else   //ceci est un autre commentaire
wwww; while dffg hh do
if dhdhdh then  dfdf else
            begin
                    b:="aaaaa"+"bbbbbb"+'c';
                        end end;
end; end.
```

Deux saisies différentes

Nous remarquons que nous sommes en face de deux saisies différentes du même texte et qu'aucune des deux ne fait apparaître la logique d'écriture par bloc du texte.
On souhaite alors construire un logiciel permettant une présentation du code fondée sur l'indentation (opération de mise en retrait de lignes selon des blocs logiques afin de présenter une meilleure lisibilité du code). La présentation restant un choix non figé, nous adopterons notre propre style d'indentation, le lecteur pourra se servir des outils fournis par la suite dans le logiciel pour changer certaines dispositions de présentation.

Une seule présentation

Nous adoptons le principe que les textes de code en général sont composés de mots qui sont soit des marqueurs syntaxiques, soit des marqueurs sémantiques, soit des marqueurs de séparation et des mots ordinaires. Les développeurs ont adopté comme présentation synthétique, la structure de "peigne" . Cette présentation est une combinaison de la "justification à gauche" d'un texte et de l'utilisation de tabulation dès qu"un bloc est ouvert ou fermé selon le schéma ci-dessous (les lignes horizontales figurent le texte, les pointillés verticaux les retraits de tabulation) :



Selon cette disposition, notre editeur de code devra pouvoir représenter les deux saisies précédentes du même texte de le même façon, soit comme ci-dessous (les mots clefs ont été surlignés en **noir gras** pour mieux faire ressortir l'indentation des lignes) :

```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin
   if x < 5 then
     y := y-7
   else
     while x in [a..b] do
     begin
       if expression(x,y)<'x' then
         for x:=1 to 15 do
           if expression(x,y)<'x' then
             if expression(x,y)<'x' then
             begin
               x:=4;
             end
             else
               if expression(x,y)<'x' then
```

```
            begin
              x:=4;
            end
        if expression(x,y)<'x' then
          a:="bonjour"      //ceci est un commentaire
        else   //ceci est un autre commentaire
          wwww;
        while dffg hh do
          if dhdhdh then
            dfdf
          else
          begin
            b:="aaaaa"+"bbbbbb"+'c';
          end
      end;
    end;
```

Spécifications de base du logiciel - Analyse et recherche des schémas d'indentation de base d'une unit Delphi

Identification des tâches pour la présentation d'un texte

Nous observons dans différents documents écrits manuellement ou par éditeur de code déjà existant qu'un certain nombre de règles non-écrites subordonnent la présentation de lignes de code. Nous remarquons que la structuration du langage influence d'une manière importante la présentation , en particulier pour les langages à structures de blocs où l'on pratique l'indentation (retrait d'une ligne par rapport à la précédente pour indiquer le début d'un nouveau bloc). Nous remarquons aussi qu'un certain nombre de styles de présentation sont laissés en libre choix.

Nous allons donc construire un logiciel pour langages structurés et plus particulièrement pour lignes de code Delphi, et nous adoptons l'idée que nos lignes de code doivent être le plus courtes possibles afin de ne pas surcharger leur lisibilité : nous priviligérons donc la briéveté par rapport au nombre plus important de lignes.

Nous postulons que la présentation du texte Delphi de 8 lignes suivant :

```
implementation
procedure ClA1.Meth(var x:integer; y:real);
begin if x < 5 then y := y-7 else     while x in [a..b] do
begin if expression(x,y)<'x' then for x:=1 to 15 do   if expression(x,y)<'x' then    if expression(x,y)<'x' then begin
x:=4;
end else    if expression(x,y)<'x' then begin x:=4; end
if expression(x,y)<'x' then  a:="bonjour"     //ceci est un commentaire
else   //ceci est un autre commentaire
end; end.
```

est "moins" lisible que le même texte beaucoup plus long (25 lignes) et réarrangé suivant (nous avons figuré avec des couleurs en gras les alignements de ligne):

```
implementation
```

```
procedure ClA1.Meth(var x:integer; y:real);
begin
  if x < 5 then
    y := y-7
  else
    while x in [a..b] do
    begin
      if expression(x,y)<'x' then
        for x:=1 to 15 do
          if expression(x,y)<'x' then
            if expression(x,y)<'x' then
            begin
              x:=4;
            end
            else
              if expression(x,y)<'x' then
              begin
                x:=4;
              end
      if expression(x,y)<'x' then
        a:="bonjour"     //ceci est un commentaire
      else   //ceci est un autre commentaire
    end;
end.
```

Principe adopté :

une ligne contient une instruction du langage.

Nous devons alors relever tous les schémas prévisibles de code et préciser le modèle de présentation que nous souhaitons leur voir suivre.

L'indentation d'une ligne

Nous repertorions ci-après les principales configurations de code possibles en partie gauche et leur comportement d'indentation souhaité en partie droite.

Imbrication de blocs

```
begin
begin
begin
begin
end
end
end
end
```

```
begin
  begin
    begin
      begin

      end
    end
  end
end
```

Lignes dans un bloc

```
begin
x:=58;
y:='kkkkkk';
z:=-14.57
end
```

```
begin
  x:=58;
  y:='kkkkkk';
  z:=-14.57
end
```

Instructions structurées

sans défaut de fermeture (repeat...until, try...except...end) :

```
begin
repeat
x:=58;
y:='kkkkkk';
z:=-14.57
until  (x> 76)and(g or not (x<7))  ;
end
```

```
begin
  repeat
    x:=58;
    y:='kkkkkk';
    z:=-14.57
  until  (x> 76)and(g or not (x<7))  ;
end
```

```
begin
try
x:=5
except
y:=x;
free;
end
end
```

```
begin
  try
    x:=5
  except
    y:=x;
    free;
  end
end
```

Instructions d'itérations

avec défaut de fermeture (for... , while....) :

```
begin
x:=26;
for i:=12 downto 5 do
x:=x+i;
y:=x-10;
end;
```

```
begin
  x:=26;
  for i:=12 downto 5 do
    x:=x+i;
  y:=x-10;
end;
```

```
begin
x:=26;
while(x<100) do
x:=x+i;
y:=x-10;
end;
```

```
begin
  x:=26;
  while(x<100) do
    x:=x+i;
  y:=x-10;
end;
```

Instructions conditonnelles

avec défaut de fermeture (if...else) :

```
begin
x:=26;
if x<100 then
x:=x+i
else
x:=x-2
y:=x-10;
end;
```

```
begin
  x:=26;
  if x<100 then
    x:=x+i
  else
    x:=x-2
  y:=x-10;
end;
```

Instructions conditionnelles imbriquées

(if...if...else...else) :

```
begin
if x<100 then
if x<200 then
if x<300 then
if x<400 then
if x<500 then
if x<600 then
x:=x+i
else
x:=x-2
y:=x-10;
end;
```

```
begin
  if x<100 then
    if x<200 then
      if x<300 then
        if x<400 then
          if x<500 then
            if x<600 then
              x:=x+i
            else
              x:=x-2
  y:=x-10;
end;
```

Mélange d'instructions à défaut de fermeture imbriquées

(rattachement du else pendant) :

```
begin
if x<100 then
if x<200 then
if x<300 then
if x<400 then
for x:=1 to 500 do
for x:=1 to 600 do
while x<700 do
if x<800 then
x:=x+i
else
x:=x-2
else
Autre(5);
y:=x-10;
end;
```

```
begin
  if x<100 then
    if x<200 then
      if x<300 then
        if x<400 then
          for x:=1 to 500 do
            for x:=1 to 600 do
              while x<700 do
                if x<800 then
                  x:=x+i
                else
                  x:=x-2
        else
          Autre(5);
  y:=x-10;
end;
```

Le découpage d'une ligne

Nous voulons que notre logiciel possède une certaine "intelligence" du texte et qu'il soit capable de découper automatiquement selon des modèles préétablis une ligne longue en plusieurs lignes. Par exemple, les 5 lignes de code de gauche devront se trouver transformées par l'éditeur de code en les 12 lignes de droite :

```
begin if x < 5 then y := y-7 else
        while x in [a..b] do
    begin if expression(x,y)<'x' then for x:=1 to 15
do
  if expression(x,y)<'x' then
          if expression(x,y)<'x' then begin x:=4;
```

```
begin
  if x < 5 then
    y := y-7
  else
    while x in [a..b] do
    begin
      if expression(x,y)<'x' then
        for x:=1 to 15 do
          if expression(x,y)<'x' then
            if expression(x,y)<'x' then
            begin
              x:=4:
```

Cette intelligence relative nécessite de fournir au logiciel des informations sur des **marqueurs sémantiques de découpage** d'une phrase. Prenons la phrase de code extraite du texte précédent :

```
begin if expression(x,y)<'x' then for x:=1 to 15 do
```

il est évident que le mot **begin** qui est déjà un mot clef du langage est aussi un marqueur sémantique de découpage, il en est de même pour les mots clefs **then** et **do** :

begin if expression(x,y)<'x' **then** for x:=1 to 15 **do**

Après la reconnaissance de ces marqueurs, le logiciel produira 3 lignes d'une seule instruction à partir de cette phrase :

begin
if expression(x,y)<'x' **then**
for x:=1 to 15 **do**

Spécifications opérationnelles du logiciel - Les grandes fonctions du logiciel et la méthodologie opératoire adoptée.

Mise en place de la présentation d'un texte

Nous posons comme postulat que la présentation de notre code résultera de deux fonctions spécifiques du logiciel ceci afin de bien séparer les actions de **découpage** du texte et celle **d'indentation** proprement dite, nous rajouterons une troisième fonctionnalité plus tard : la coloration syntaxique des mots clefs du langage.

Méthode de découpage d'une ligne

Nous proposons de construire une méthode **CouperLigne** qui balaie entièrement toutes les lignes du texte dans un premier passage. A cette méthode **CouperLigne** nous attribuons deux fonctions :

Fonction découpage proprement dit, c'est la méthode **CouperLigne** qui effectuera la reconnaissance des marqueurs de découpage et qui réinserera immédiatement les nouvelles lignes générées, dans le texte comme dans l'exemple ci-dessous:

begin if x > 46 **then** for x:=46 downto 15 **do** y:=y+x **;** z:=y-1 ;

la méthode **CouperLigne** doit reconnaître les 4 marqueurs de découpage : **begin** , **then** , **do** et **;**

Après la reconnaissance de ces marqueurs la méthode **CouperLigne** doit insérer les 5 lignes ci-dessous dans le texte à la place de la ligne précédente :

```
begin
if x > 46 then
for x:=46 downto 15 do
y:=y+x ;
z:=y-1 ;
```

L'ensemble de tous les marqueurs de découpage est stocké dans une structure de données à accès direct afin d'être accessible immédiatement (un ensemble set of en Delphi), nous la noterons EnsMotDeCoupe.

Nous remarquons que la méthode **CouperLigne** doit pouvoir découper une ligne contenant un nombre quelconque et non connu à l'avance de marqueurs de découpage :

La méthode **CouperLigne** doit découper aussi bien la ligne :

begin if x > 46 **then** for x:=46 downto 15 **do** y:=y+x **;** z:=y-1 ;

que découper la ligne :

begin x:=5

Nous proposons de construire une méthode récursive qui va découper une ligne de gauche à droite en deux lignes, puis se rappeler récursivement sur la deuxième ligne jusqu'à épuisement des marqueurs de découpage. Prenons un exemple :

Soit la ligne de code suivante où pour des raisons de compréhension nous avons fait figurer son numéro dans la liste des lignes (ici ce serait la 17 ème ligne du texte) :

n° 17 - if x=0 then x:= 1 ; y:=5 begin

La méthode **CouperLigne** reconnaît le marqueur **then** elle scinde donc la ligne n°17 en 2 morceaux :

n° 17 - [if x=0 then] [x:= 1 ; y:=5 begin]

Puis la méthode **CouperLigne** remplace la ligne originale n°17 par deux nouvelles lignes n°17 et n°18 obtenues par découpage :

n° 17 - [if x=0 then] *n° 18* - [x:= 1 ; y:=5 begin]

Enfin la méthode **CouperLigne** se rappelle sur la ligne n°18 où elle reconnaît le marqueur **;** :

n° 18 - [x:= 1 ;] [y:=5 begin]

Elle réengendre 2 nouvelles lignes

n° 18 - x:= 1 ;
n° 19 - y:=5 begin

La méthode **CouperLigne** se rappelle une dernière fois récursivement sur la ligne n°19 et repère le marqueur **begin** , pour fournir finalement le nouveau texte suivant :

n° 17 - if x=0 then
n° 18 - x:= 1 ;
n° 19 - y:=5
n° 20 - begin

Traitement des commentaires

Nous savons par expérience que le code doit être documenté à l'aide de commentaires, l'attitude des développeurs est d'utiliser soit un commentaire mono-ligne pour une information brève et ciblée sur une ligne de code particulière, soit un commentaire plus long de plusieurs lignes que nous dénommerons commentaire multi-lignes. La méthode **CouperLigne** doit respecter les commentaires en particulier les commentaires mono-ligne.

1)En effet la ligne de code Delphi suivante :

n° 17 - if x=0 then // ceci est un commentaire

ne doit pas être découpée en les 2 lignes suivantes :
n° 17 - if x=0 then
n° 18 -// ceci est un commentaire

2)En revanche la ligne :

n° 17 - if x=0 then x:= 1 ; y:=5 // ceci est un commentaire

sera découpée en :
n° 17 - if x=0 then
n° 18 - x:= 1 ;
n° 19 - y:=5 // ceci est un commentaire

Nous normalisons la présentation des commentaires multi-lignes afin d'avoir une représentation standard dans tous les textes,en Delphi :

```
{  Les nombres avec un séparateur décimal
   ou un exposant désignent des réels, alors que
   les autres nombres désignent des entiers.
}
```

Entre un marqueur syntaxique de début de commentaire multi-lignes ('**{**'en Delphi) et un marqueur de fin de commentaire multi-lignes ('**}**'en Delphi), les lignes de commentaires ne subiront pas de présentation particulière, et elles ne seront ni découpées, ni indentées.

Remarque importante :

Les mots d'une ligne de code peuvent être différenciés selon leur mode d'interprétation soit rien (aucune action, c'est alors un mot banal), soit c'est un marqueur syntaxique (début de bloc, fin de bloc,...), soit c'est un marqueur sématique de découpage, soit c'est un mot clef (utilisable pour la coloration syntaxique des mots clefs). Certains éléments peuvent être à la fois marqueur syntaxique , marqueur de découpage etc.. (comme par exemple le mot clef begin). Nous englobons toutes ces catégories sous un seul vocable: les catégories lexicales que nous dénoterons **CategLexeme**.

Voici ci-dessous les catégories lexicales **CategLexeme** qui ont été utilisées dans le logiciel construit :

isRien, isStartBloc, isEndBloc ,isParagraphe, isAlinea, isTeteBloc, isMilieuInstrStruct, isDebutDefautFermeture, isFinDeLigne, isMilieuDefautFermeture, isDebutDefautFermetureCompos, isDebutInstrStruct, isEndInstrStruct, IsStartComment, IsEndComment, IsComment, isChaine, isDebutParagrapheTry, isMilieuParagrapheTry, isAsLignePrec, isDelimit, isVide .

Nous construirons une méthode qui permet d'extraire les mots d'une ligne de code et une méthode qui fournit la catégorie lexical d'un mot, nous aurons ainsi à notre disposition un petit analyseur lexical pour l'indentation (et le coloriage)

Stockage des information collectées

Nous assignons aussi à la méthode **CouperLigne** un travail de conservation dans une liste des informations collectées sur une ligne. Sachant que la prochaine activité d'indentation du logiciel necessite pour une ligne donnée de connaître le genre de ligne de la précédente nous stockons dans une liste **ListeFirstKeyLine** un lexème (la catégorie lexicale du premier mot de la ligne ainsi que le mot lui-même).

Ainsi, en reprenant l'exemple précédent après action de la méthode **CouperLigne** sur la ligne
n° 17 - if x=0 then x:= 1 ; y:=5 // ceci est un commentaire

on obtient un nouveau texte :

....
n° 17 - **if** x=0 then
n° 18 - **x** := 1 ;
n° 19 - **y** :=5
n° 20 - **begin**

et une liste de lexèmes **ListeFirstKeyLine** qui s'agrandit de 4 éléments :

.....
n° 17 - (**if** , isDebutDefautFermetureCompos)
n° 18 - (**x** , isRien)
n° 19 - (**y** , isRien)
n° 20 - (**begin** , isStartBloc)

Moteur de décisions

Nous utilisons pour prendre les décisions de présentation d'une ligne de code, un moteur de décision fondé sur **un automate à pile de mémoire**.

L'analyse et la construction se font ligne à ligne les décisions d'actions sont dirigées par un automate à pile qui permet pour chaque ligne de savoir dans quel contexte la ligne se situe par connaissancede la catégorie du premier lexème de la ligne (état de l'automate) et par consultation d'une pile contextuelle dénotée **PileBlocs,** contenant des informations sur les blocs imbriqués et sur les instructions à défaut de fermeture imbriquées dans les blocs.

Nous rappellons que certains langages comme Delphi, Java, C++, C# etc... possèdent des instructions structurées avec un défaut de fermeture (pas de marqueur syntaxique de fin de l'instruction), Visual Basic quant à lui ne possède pas d'instruction à défaut de fermeture.

Par exemple l'instruction de boucle **while** fermée en VB et non fermée dans les autres langages Delphi, Java, C++, C# :

En Delphi :
while x < 5 **do**
x := x+2
	En Delphi :
	while x < 5 **do**
	begin
	x := x+2;
	y:=x-1;
	end

En C++, Java, C# :
while (w < 5)
x +=2;
	En C++, Java, C# :
	while (w < 5)
	{
	x += 2;
	y = x-1;
	}

En VB : **while** x < 5 x = x+2 **wend**	En VB : **while** x < 5 x = x+2 y = x-1 **wend**

Notre moteur de décisions doit donc être capable dans un langage comme Delphi de proposer une indentation particulière pour de telles instructions. En reprenant l'exemple de l'instruction **while**, nous aurons :

texte de la ligne avant indentation	texte de la ligne après indentation

while x < 5 **do**
begin
x := x+2;
y:=x-1;
end
	while x < 5 **do**
	begin
	  x := x+2;
	  y:=x-1;
	end

while x < 5 **do**
x := x+2;
y:=x-1;
	while x < 5 **do**
	  x := x+2;
	y:=x-1;

Dans le cas où plusieurs instructions de ce type sont imbriquées, le moteur de décision doit être capable de situer correctement la dernière ligne qui "ferme" les imbrications et de rattacher la prochaine instruction au bon bloc.

Exemple :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
while x < 7 **do**
while x < 6 **do**
while x < 5 **do**
begin
x := x+2;
y:=x-1;
end
	for i:=1 to 10 **do**
	  while x < 7 **do**
	  while x < 6 **do**
	  while x < 5 **do**
	  begin
	  x := x+2;
	  y:=x-1;
	  end

for i:=1 to 10 **do**
while x < 7 **do**
while x < 6 **do**
while x < 5 **do**
x := x+2;
y:=x-1;
	for i:=1 to 10 **do**
	  while x < 7 **do**
	  while x < 6 **do**
	  while x < 5 **do**
	  x := x+2;
	y :=x-1;

Pour ce faire l'automate de décision travaille avec la pile contextuelle **PileBlocs** dans laquelle il range les ouvertures de blocs structurés et les rangs d'indentation à chaque fois qu'il rencontre une instruction à défaut de fermeture imbriquée (lexèmes : isDebutDefautFermeture et isDebutDefautFermetureCompos).

Enfin, dans la catégorie des instructions à défaut de fermeture, le **if...else** prend une place particulière due au problème de la résolution classique du **else** pendant (nous attribuons au **if** une categorie lexicale spéciale isDebutDefautFermetureCompos et au **else** la catégorie isMilieuDefautFermeture) :

if P1 **then if** P2 **then else** A **else** B

Problème d'ambiguïté résolue classiquement par le compilateur par rattachement du premier 'else' au 'if' le plus proche. Notre logiciel doit tenir compte de cette démarche et fournir une présentation adéquate qui prenne en compte ce rattachement au if le plus proche :

```
if P1 then if P2 then else A else B  ⇒  if P1 then
                                          if P2 then
                                          else A
                                        else B
```

Notre logiciel doit pouvoir interpréter et indenter aisément des situations comme celle qui suit (les couleurs ne sont mises que pour bien faire ressortir les alignements des lignes) :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
if y<4 **then**
if y<3 **then**
while x < 7 **do**
if y<2 **then**
while x < 6 **do**
while x < 5 **do**
x := x+2
else
y := x-1
else
z := 8 ;
	for i:=1 to 10 **do**
	if y<4 **then**
	if y<3 **then**
	while x < 7 **do**
	if y<2 **then**
	while x < 6 **do**
	while x < 5 **do**
	x := x+2
	else
	y := x-1
	else
	z := 8 ;

Les piles contextuelles

Nous utilisons une pile auxiliaire de contexte que nous notons **PileIndent,** qui contient uniquement les valeurs des retraits de chaque début de bloc. L'ensemble des deux piles permet à l'automate d'avoir une **information constante** sur la valeur du retrait à attribuer à une ligne à l'intérieur d'un bloc. Les piles **PileIndent** et **PileBlocs** sont liées logiquement entre elles selon le schéma d'information ci-dessous :

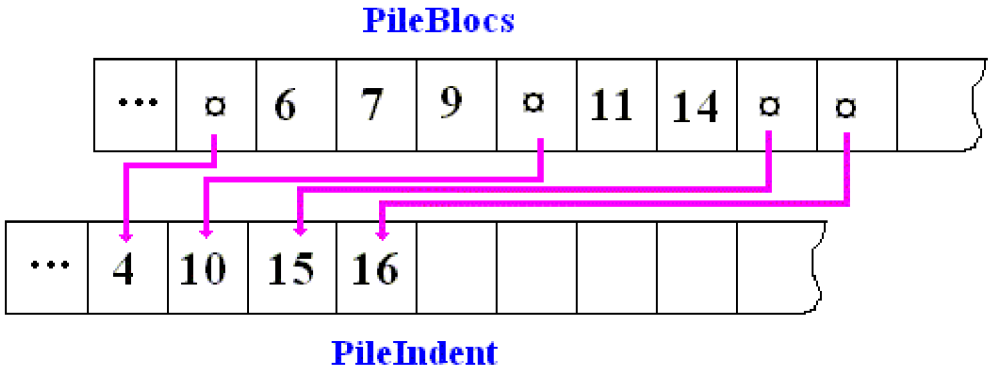
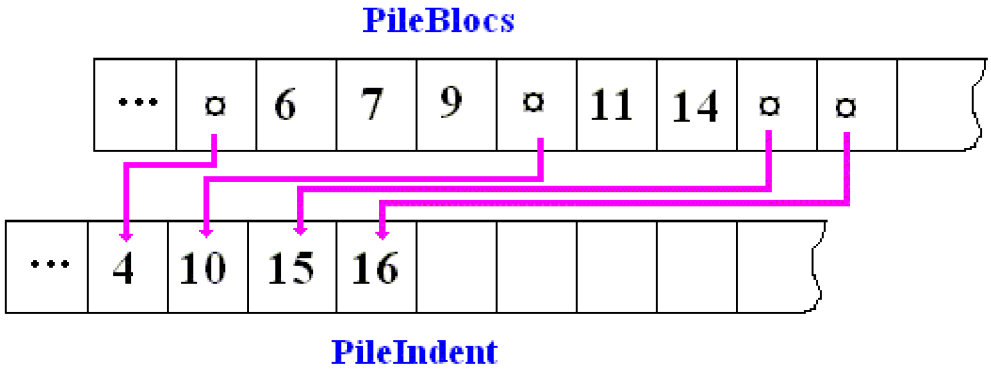


Schéma de piles pouvant correspondre aux lignes de code suivantes :

lignes abstraites	lignes en Delphi
< ... , isStartBloc >	Retrait=4 / begin
< ... , isDebutDefautFermetureCompos >	Retrait =6 / if .... then
< ... , isDebutDefautFermetureCompos >	Retrait =7 / if .... then
< ... , isDebutDefautFermeture >	Retrait =8 / for ... to
< ... , isDebutDefautFermetureCompos >	Retrait =9 / if .... then
< ... , isStartBloc >	Retrait =10 / begin
< ... , isDebutDefautFermetureCompos >	Retrait =11 / if .... then
< ... , isDebutDefautFermeture >	Retrait =12 / for ... to
< ... , isDebutDefautFermeture >	Retrait =13 / while ... do
< ... , isDebutDefautFermetureCompos >	Retrait =14 / if .... then
< ... , isStartBloc >	Retrait =15 / begin
< ... , isRien >	Retrait =16 / x := 47
< ... , isStartBloc >	Retrait =16 / begin



Empilement :

> Lorsqu'une marque de début de bloc ⊗ est stockée dans la **PileBlocs** , la valeur actuelle de la position du retrait d'indentation est stockée dans la pile auxiliaire **PileIndent**.

Dépilement :

> A chaque fois qu'une marque de début de bloc ⊗ est dépilée de la pile **PileBlocs** , la pile **PileIndent** est dépilée parallèlement de son sommet de telle façon que le sommet de **PileIndent** représente toujours le niveau d'indentation du bloc en cours.

Ainsi les deux piles **PileBlocs** et **PileIndent** représentent une vue abstraite de tout le texte sous l'angle de la présentation. Elles sont utilisées par un automate de décision qui les remplit, les consulte et les modifie en fonction de situations spécifiques.

Les différentes fonctions de l'automate décisionnel

Méthodes de base

Il nous faut donc construire nos outils de base permettant ces différents types de positionnement. Nous listons ci-dessous les méthodes de notre classe éditeur qui travaillent sur une ligne de texte et que nous avons classées pour des raisons pédagogiques, par catégories :

```
{ Méthodes de positionnement d'une ou plusieurs lignes }
function  GetPosIndent(NumLigne:integer):integer;
function FirstChar(NumLigne:integer):integer;
function NbrBlancs(NumLigne:integer):integer;
function Decalage(nbr:integer):string;
function DecalageTAB(nbrTab:integer):string;
procedure AjusteLigneNextTAB(NumLigne:integer);
procedure AjusteLigneRight(NumLigne:integer);
procedure AvancerLigne(NumLigne,Nbrtab:integer);
procedure ReculerLigne(NumLigne,Nbrtab:integer);
procedure PositionneSurLignePrec(NumLigne:integer);
procedure PositionneLigneSurIndent(NumLigne:integer;Depiler:boolean);
procedure PositionneLigneDeFinDefautFermerure(NumLigne:integer);
procedure PositionneLigneSurLastDefautFermeture(NumLigne:integer);
procedure AlignerPargrf(Debut,Fin:integer;tabuler:boolean);
procedure AvancerPargrf(Debut,Fin:integer);
procedure ReculerPargrf(Debut,Fin:integer);

{ Gestion des piles }
procedure RAZPileIndent;
procedure RAZPileBlocs;
procedure DepileNextMarkBloc;

{ Analyseur lexical }
procedure ExtraitMot(Ligne:String; var numcar:integer; var Mot:string;  var IsMot: MotsLimites;var
TypeBloc:CategLexeme);

{ Découpage d'une ligne }
procedure StockKeyLine(NumLigne:integer);
procedure CouperLigne(NumLigne:integer);

{ Indentation d'une ligne : Moteur de décision }
procedure IndentLigneIfMotNotRien(NumLigne:integer;Motactuel,Motprec:string;EtatMot,EtatmotPrec:
CategLexeme);
procedure IndentLigneIfMotisRien(NumLigne:integer;Motactuel,Motprec:string;EtatMot,EtatmotPrec:
CategLexeme);
procedure IndentUneLigne(NumLigne:integer;indentTout:boolean);

{ Analyse et indentation de tout le texte }
procedure IndenterTout;
procedure AnalyseComplete;
```

Toutes ces méthodes participent aux actions de découpage et d'indentation proprement dite de l'automate. Nous décrivons ci-après les deux actions.

Découpe d'une ligne

Eléments utiles au découpage d'une ligne :

- Méthode ExtraitMot (Ligne, rang, Mot, MotIs, MotIsBloc) est l'analyseur lexical qui extrait consécutivement dans une ligne les mots (dans la variable Mot) et leur catégorie lexicale (dans la variable MotIsBloc).
- Méthode StockKeyLine(NumLigne:integer) range le premier mot de la ligne et sa catégorie lexicale dans une liste nommée ListeFirstKeyLine.
- Rappelons enfin que toutes les catégories lexicales sont décrites dans le type **CategLexeme** :

> isRien, isStartBloc, isEndBloc ,isParagraphe, isAlinea, isTeteBloc, isMilieuInstrStruct, isDebutDefautFermeture, isFinDeLigne, isMilieuDefautFermeture, isDebutDefautFermetureCompos, isDebutInstrStruct, isEndInstrStruct, IsStartComment, IsEndComment, IsComment, isChaine, isDebutParagrapheTry, isMilieuParagrapheTry, isAsLignePrec, isDelimit, isVide .

Nous ne rentrerons pas dans le détail du code, seulement dans les grandes lignes structurelles afin de comprendre comment l'automate prend ses décisions voici en texte algorithmique :

```
Algorithme CouperLigne
entrée : NumLigne  ∈  entier
...
EnsMotdeCoupe = { isFinDeLigne, isMilieuInstrStruct, isStartBloc, isEndInstrStruct,
isMilieuDefautFermeture, isEndBloc, isTeteBloc, isStartComment, isEndComment, isComment }

début
 Tantque ( non fin de la ligne ) et ( MotIsBloc  ∉  EnsMotdeCoupe ) faire
    ...
    ExtraitMot ( Ligne, rang, Mot, MotIs, MotIsBloc );
    ...
    si MotIsBloc ∈ {IsComment, isMilieuParagrapheTry, isTeteBloc}alors
      StockKeyLine ( NumLigne );
      sortir de la boucle;
   fsi ;
    si MotIsBloc ∈ {isStartBloc, isMilieuInstrStruct, isMilieuDefautFermeture,
            isEndInstrStruct, isEndBloc, IsStartComment, IsEndComment} alors
      Traiter les cas de découpage de la ligne et des commentaires
    sinon
    si (MotIsBloc ∈ {isFinDeLigne} ) et (non ChaineEnCours) alors
     ligne à décomposer et ajouter les nouvelles lignes dans le texte
    fsi
  fsi
 fTant
 StockKeyLine ( NumLigne );
fin-CouperLigne
```

Exemple :

Afin de bien comprendre ce que fait ce premier traitement sur le texte prenons le texte Delphi de 6 lignes suivant :

```
for i:=1 to 10 do if y<4 then
begin  if y<3 then while x < 7 do begin
if y<2 then  while x < 6 do while x < 5 do
x := x+2 else y := x-1
end end
else z := 8 ;
```

Appliquons 6 fois (une fois pour chaque ligne) la méthode CouperLigne (en rouge souligné le premier marqueur de découpage repéré par la méthode)

Entrée de la méthode **CouperLigne** : une ligne de code.	sortie de **CouperLigne** : une ou plusieurs lignes.	Ajout dans la liste ListeFirstKeyLine
for i:=1 to 10 **do** **if** y<4 **then**	**for** i:=1 to 10 **do** **if** y<4 **then**	< Mot = **for** , MotIsBloc >

if y<2 **then** **while** x < 6 **do while** x < 5 **do**
 	
La deuxième ligne est sécable :
while x < 6 **do while** x < 5 **do**
	if y<2 **then**
	while x < 6 **do while** x < 5 **do**
	
	par rappel récursif :
	while x < 6 **do**
	while x < 5 **do**
		< Mot = **if** , MotIsBloc >

etc...

Rappelons que MotIsBloc ∈ **CategLexeme**, voici les résultats effectifs produits :

Texte après découpage	liste ListeFirstKeyLine

for i:=1 to 10 **do**
if y<4 **then**
begin
if y<3 **then**
while x < 7 **do**
begin
if y<2 **then**
while x < 6 **do**
while x < 5 **do**
x := x+2
else
y := x-1
end
end
else
z := 8 ;
	< **for** , isDebutDefautFermeture >
	< **if** , isDebutDefautFermetureCompos >
	< **begin** , isStartBloc >
	< **if** , isDebutDefautFermetureCompos >
	< **while** , isDebutDefautFermeture >
	< **begin** , isStartBloc >
	< **if** , isDebutDefautFermetureCompos >
	< **while** , isDebutDefautFermeture >
	< **while** , isDebutDefautFermeture >
	< x , isRien >
	< **else** , isMilieuDefautFermeture >
	< y , isRien >
	< **end** , isEndBloc >
	< **end** , isEndBloc >
	< **else** , isMilieuDefautFermeture >
	< z , isRien >

Algorithmes d'indentation

L'indentation est effectuée par un nouveau passage sur le texte, pour des raisons de clarté les actions ont été divisées en 3 algorithmes interdépendants.

Algorithme IndentUneLigne
entrée : NumLigne ∈ entier
.....
Mot : premier mot de la ligne
MotIsBloc : catégorie lexicale du premier mot de la ligne

début
...
si MotIsBloc = IsStartComment **alors**
...*// c'est un début de commentaire multi-lignes*
sinon
si MotIsBloc = IsEndComment **alors**
...*//le découpage de ligne met une fin de commentMulti sur une ligne seule*
sinon
si MotIsBloc = IsChaine **alors**
...*// le début de ligne est une chaine*
sinon
si MultiEnCours **alors**
... *//on est dans un commentaire multi-lignes*
fsi
fsi
fsi
fsi
....
si MotIsBloc ∈ [{ isRien , isVide } **alors**
/ le 1er mot de la ligne n'est ni un début, ni une fin de Bloc,*
*ni un paragraphe , ou la ligne est vide */*
IndentLigneIfMotisRien *// le 1er mot de la ligne est un marqueur d'indentation*
sinon *// MotIsBloc <> isRien ou MotIsBloc <> isVide*
IndentLigneIfMotNotRien
fsi
fin-IndentUneLigne

Algorithme IndentLigneIfMotNotRien
// examine l'état lexical du premier mot de la ligne actuelle pour décider
*on examine tous les cas sauf **isRien** et **isVide***
début
selon la valeur de **EtatMot** de la ligne actuelle
IsComment, IsStartComment :
selon la valeur de **EtatmotPrec** de la ligne précédente
isStartBloc,isDebutInstrStruct,isMilieuInstrStruct,
isMilieuDefautFermeture,isDebutDefautFermeture,
isDebutDefautFermetureCompos :
Avancer la ligne d'une tabulation

isEndBloc, isEndInstrStruct, isRien :
si on est dans un bloc à défaut de fermeture **alors**
on Positionne la Ligne sur la valeur De Fin de DefautFermerure

sinon
on Positionne la Ligne Sur la Ligne Précédente
fsi

tous les autres cas :
on Positionne la Ligne Sur la Ligne Précédente

fin selon la valeur de **EtatmotPrec**

isStartBloc, isDebutInstrStruct :
si EtatmotPrec ∈ { isStartBloc, isDebutInstrStruct,
isDebutParagrapheTry, isMilieuParagrapheTry, isMilieuInstrStruct } **alors**
Avancer la ligne d'une tabulation
sinon
si EtatmotPrec ∈ { isAsLignePrec, IsEndComment, IsComment, IsStartComment } **alors**
on Positionne la Ligne Sur la Ligne Précédente
sinon
Traitement des cas des blocs à defaut fermeture
fsi
fsi
Empiler dans **PileBlocs** une marque de début de bloc ;
Empiler dans **PileIndent** la valeur de la position d'indentation actuelle

isEndBloc, isEndInstrStruct :
on dépile **PileBlocs** jusqu'à la prochaine marque de début de bloc ;
on Positionne la ligne Sur l'Indent ation actuelle ;
on examine le sommet de PileBlocs qui sert
à positionner un drapeau de déclaration ou de fin de bloc à défaut de fermeture

isParagraphe :
traitement du cas du paragraphe ;

isDebutParagrapheTry:
traitement du cas du paragraphe du type try ;

isMilieuParagrapheTry:
on Positionne la ligne Sur l'Indent ation actuelle ;

isAlinea:
traitement du cas de l'alinéa ;

isTeteBloc:
traitement du cas dela tête de bloc ;

isMilieuInstrStruct :
traitement du cas d'un milieu d'instruction structurée ;

isDebutDefautFermeture, isDebutDefautFermetureCompos :
traitement du cas des débuts de bloc à défaut de fermure;

isMilieuDefautFermeture :
si on est dans un bloc à défaut de fermeture alors
on Positionne la Ligne sur la dernière indentation d'ouverture de Defaut Fermerure
sinon
on Positionne la Ligne Sur la Ligne Précédente
fsi

isAsLignePrec :
on Positionne la Ligne Sur la Ligne Précédente

fin selon la valeur de **EtatMot**
fin-IndentLigneIfMotNotRien

Algorithme IndentLigneIfMotisRien
*// l'état lexical du premier mot de la ligne actuelle est **isRien** ou **isVide***
on examinel'état lexical du premier mot de la ligne précédente
début
selon la valeur de **EtatmotPrec** de la ligne précédente
isStartBloc, isDebutInstrStruct, isParagraphe,
isAlinea, isTeteBloc, *isMilieuInstrStruct, isMilieuParagrapheTry* :
Avancer la ligne d'une tabulation

IsStartComment, isVide, IsEndComment, IsComment :
on Positionne la Ligne Sur la Ligne Précédente

isDebutDefautFermeture, isDebutDefautFermetureCompos, isMilieuDefautFermeture :
Avancer la ligne d'une tabulation

isRien,isEndBloc, isEndInstrStruct :
si on est dans un bloc à défaut de fermeture **alors**
on Positionne la Ligne sur la valeur De Fin de DefautFermerure
sinon
on Positionne la Ligne Sur la Ligne Précédente
fsi
fin selon la valeur de **EtatmotPrec**
fin-IndentLigneIfMotisRien

Les piles contextuelles

Après avoir produit et **normalisé** un nouveau texte et la liste des premiers lexèmes, l'automate a repris le nouveau texte **normalisé** pour l'indenter à l'aide des algorithmes précédents.

Nous avons juste mentionné la gestion des piles qui est essentielle aux bonnes prises de décision. Afin de cerner de plus près l'intérêt de ces piles , nous reprenons l'exemple donné au paragraphe des spécifications opérationnelles, puis nous le modifierons :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
if y<4 **then**
if y<3 **then**
while x < 7 **do**
if y<2 **then**
while x < 6 **do**
while x < 5 **do**
x := x+2
else
y := x-1
else
z := 8 ;
	for i:=1 to 10 **do**
	if y<4 **then**
	if y<3 **then**
	while x < 7 **do**
	if y<2 **then**
	while x < 6 **do**
	while x < 5 **do**
	x := x+2
	else
	y := x-1
	else
	z := 8 ;

La gestion de la pile contextuelle **PileBlocs** est alors essentielle dans ce genre d'éventualité : l'automate empile la valeur actuelle de retrait de chaque lexème du type isDebutDefautFermetureCompos, l'automate dépile à chaque lexème de catégorie isMilieuDefautFermeture.

La pile **PileBlocs** contient uniquement des symboles de **marquage de début de bloc** ⊗ et de **rang d'indentation d'une ligne de catégorie à défaut fermeture** :

Reprenons l'exemple précédent et visualisons l'évolution prévue de la pile **PileBlocs** :

for i:=1 to 10 **do**	[.....] rien n'est empilé (retrait actuel=1)
if y<4 **then**	[....., **2**] retrait actuel=2 empilé
if y<3 **then**	[....., 2 , **3**] retrait actuel=3 empilé
while x < 7 **do**	[....., 2 , 3] rien n'est empilé (retrait actuel=4)
if y<2 **then**	[....., 2 , 3 , **5**] retrait actuel=5 empilé
while x < 6 **do**	[....., 2 , 3 , 5] rien n'est empilé (retrait actuel=6)
while x < 5 **do**	[....., 2 , 3 , 5] rien n'est empilé (retrait actuel=7)
x := x+2	[....., 2 , 3 , 5] rien n'est empilé (retrait actuel=7) décalage +1 par rapport à la ligne précedente
else	[....., 2 , 3] **5** est dépilé (retrait actuel=5)
y := x-1	[....., 2 , 3] rien n'est empilé (retrait actuel=5) décalage +1 par rapport à la ligne précedente
else	[....., 2] **3** est dépilé (retrait actuel=3)
z := 8 ;	[....., 2] rien n'est empilé (retrait actuel=3) décalage +1 par rapport à la ligne précedente

Si la prochaine ligne est un **else** (isMilieuDefautFermeture) le sommet de pile (valeur=2) indique le bon retrait, sinon la pile contextuelle **PileBlocs** n'est pas consultée et c'est une autre procédure de décision qui est mise en oeuvre. Nous n'avons pas fait figurer la pile **PileIndent**, car celle-ci n'a pas évolué dans ce morceau de code à indenter du fait qu'aucun nouveau bloc n'a été ouvert. **PileIndent** contient dès le départ la valeur 0, soit la valeur du premier retrait du texte.

Introduisons des blocs dans ce texte :

texte de la ligne avant indentation	texte de la ligne après indentation

for i:=1 to 10 **do**
if y<4 **then**
begin
if y<3 **then**
	for i:=1 to 10 **do**
	  if y<4 **then**
	  begin
	  if y<3 **then**

while x < 7 **do**
begin
if y<2 **then**
while x < 6 **do**
while x < 5 **do**
x := x+2
else
y := x-1
end
end
else
z := 8 ;
	while x < 7 **do**
	begin
	if y<2 **then**
	while x < 6 **do**
	while x < 5 **do**
	x := x+2
	else
	y := x-1
	end
	end
	else
	z := 8 ;

Visualisons sous la forme de deux tableaux, la nouvelle évolution prévue de la pile **PileBlocs** (⊗ = marque de début de bloc) et celle de la pile **PileIndent** :

for i:=1 to 10 **do**	**PileBlocs** = [.....] rien n'est empilé (retrait actuel=1) **PileIndent** = [.....]
if y<4 **then**	**PileBlocs** = [....., **2**] 2 empilé retrait actuel=2 **PileIndent** = [.....]
begin	**PileBlocs** = [....., 2 , ⊗] marque de bloc empilée (retrait actuel=2) **PileIndent** = [..... , 2]
if y<3 **then**	**PileBlocs** = [....., 2 , ⊗ , 3] 3 empilé retrait actuel=3 **PileIndent** = [..... , 2]
while x < 7 **do**	**PileBlocs** = [....., 2 , ⊗ , 3] rien n'est empilé (retrait actuel = **4**) **PileIndent** = [..... , 2]
begin	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗] marque de bloc empilée (retrait actuel = **4**) **PileIndent** = [..... , 2 , 4]
if y<2 **then**	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , **5**] 5 empilé retrait actuel=5 **PileIndent** = [..... , 2 , 4]
while x < 6 **do**	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , 5] rien n'est empilé (retrait actuel=6) **PileIndent** = [..... , 2 , 4]
while x < 5 **do**	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , 5] rien n'est empilé (retrait actuel=7) **PileIndent** = [..... , 2 , 4]
x := x+2	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗ , 5] rien n'est empilé (retrait actuel=8) décalage +1 par rapport à la ligne précedente

	PileIndent = [..... , 2 , 4]
else	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗] **5** est dépilé (retrait actuel=5) **PileIndent** = [..... , 2 , 4]
y := x-1	**PileBlocs** = [....., 2 , ⊗ , 3 , ⊗] rien n'est empilé (retrait actuel=6) décalage +1 par rapport à la ligne précedente **PileIndent** = [..... , 2 , 4]
end	**PileBlocs** = [....., 2 , ⊗ , 3] ⊗ est dépilé (retrait actuel = 4) **PileIndent** = [..... , 2] , 4 est dépilé
end	**PileBlocs** = [....., 2] ⊗ **, 3 sont dépilés** (retrait actuel=2) **PileIndent** = [.....] , 2 est dépilé
else	**PileBlocs** = [.....] **2** est dépilé (retrait actuel=2)
z := 8 ;	**PileBlocs** = [.....] rien n'est empilé (retrait actuel=3)

[retrait] = Valeur du curseur d'indentation	**PileBlocs**	**PileIndent**
[1] **for** i:=1 to 10 **do**	[.....]	[.....]
[2] **if** y<4 **then**	[..... , 2]	[.....]
[2] **begin**	[....., 2 , ⊗]	[..... , 2]
[3] **if** y<3 **then**	[....., 2 , ⊗ , 3]	[..... , 2]
[4] **while** x < 7 **do**	[....., 2 , ⊗ , 3]	[..... , 2]
[4] **begin**	[....., 2 , ⊗ , 3 , ⊗]	[..... , 2 , 4]
[5] **if** y<2 **then**	[....., 2 , ⊗ , 3 , ⊗ , 5]	[..... , 2 , 4]
[6] **while** x < 6 **do**	[....., 2 , ⊗ , 3 , ⊗ , 5]	[..... , 2 , 4]
[7] **while** x < 5 **do**	[....., 2 , ⊗ , 3 , ⊗ , 5]	[..... , 2 , 4]
[8] x := x+2	[....., 2 , ⊗ , 3 , ⊗ , 5]	[..... , 2 , 4]
[5] **else**	[....., 2 , ⊗ , 3 , ⊗]	[..... , 2 , 4]
[6] y := x-1	[....., 2 , ⊗ , 3 , ⊗]	[..... , 2 , 4]
[4] **end**	[....., 2 , ⊗ , 3]	[..... , 2]
[2] **end**	[....., 2]	[.....]
[2] **else**	[.....]	[.....]
[3] z := 8 ;	[.....]	[.....]

Nous voyons que les deux piles permettent à l'automate de connaître en permanence en **valeur absolue de tabulations**, la profondeur de retrait du bloc où se trouve la ligne à indenter, toutefois les éléments déclencheur d'un nouveau retrait d'indentation ne se réduisent pas uniquement aux ouvertures/fermetures de bloc et d'instructions du type if...then...else. En outre pour des raisons de choix de présentation une ligne peut s'aligner sur la ligne précédente ou bien plutôt avancer d'une tabulation sur la ligne précédente; dans cette dernière hypothèse nous procédons par **positionnement relatif** et non pas par positionnement absolu.

L'automate "moteur de décision" va donc agir par positionnement **absolu** dans certains cas et par positionnement **relatif** dans d'autres. C'est de la variabilité et du dosage entre ces deux types d'actions que nous obtiendrons une présentation personnalisée.

Ci-dessous l'insertion de la classe indentateur de code dans une éditeur syntaxique :

Editeur Syntaxique D:_coursinfo341\ChapProjets\EditIndent\TestsDelphi\DelphitestDesordre.pas

Fichier Edition Paramètres Version

Texte: Brut indent — Colorisation: Oui / Non — Tabulation: Tab = 1 / Tab = 2 / Tab = 3 — Auto-indent: On — Indentation - Coloris

```
begin
  for i:=1 to 15 do   x:=45;for i:=1 to 15 do  begin
    Mot:='/'+Ligne[index];IsMot:= IsComment;  end;
  if i=15 then   if i=15 then      if i=15 then          x:=45 ;
  Mot:='/'+Ligne[index];
  IsMot:= IsComment;   if i=15 then
                 if i=15 then
  x:=45       else
       IsMot:= IsComment;
    if i=15 then x:=45 else
        begin  Mot:='/'+Ligne[index];
     IsMot:= IsComment; end; if i=15 then  begin      x:=45;
  end    else    begin
     Mot:='/'+Ligne[index] ;  IsMot:= IsComment ;
 end  ; if i=15 then   begin   Mot:='/'+Ligne[index];  IsMot:= IsComment;
  end;
  while i=15 do  x:=45;
    while i=15 do   begin    Mot:='/'+Ligne[index];
       IsMot:= IsComment;      end;     end
```

Affichage du texte Delphi au format non organisé, avant lancement de l'algorithme d'indentation.

Ligne : 524 | Modifié | Insertion | Delphi

Editeur Syntaxique D:_coursinfo341\ChapProjets\EditIndent\TestsDelphi\DelphitestDesordre.pas

Fichier Edition Paramètres Version

Texte: Brut indent — Colorisation: Oui / Non — Tabulation: Tab = 1 / Tab = 2 / Tab = 3 — Auto-indent: On — Indentation - Coloris

```
begin
  for i:=1 to 15 do
    x:=45;
  for i:=1 to 15 do
  begin
    mot:='/'+ligne[index];
    ismot:= iscomment;
  end;
  if i=15 then
    if i=15 then
      if i=15 then
        x:=45 ;
  Mot:='/'+Ligne[index];
  ismot:= iscomment;
  if i=15 then
    if i=15 then
      x:=45
    else
      IsMot:= IsComment;
```

Affichage du même texte Delphi après demande d'action de l'indentation, avec une tabulation de 3 (3 blancs pour le décalage)

Ligne : 524 | Modifié | Insertion | Delphi

Exercices chapitre 6

Ex-1 : Soit G la C-grammaire suivante et L(G) le langage engendré par G :

G : V_N = { S, A ,B }
V_T = { **(** , **)** , **o** }
Axiome : A
Règles :
1 : A → **(** A
2 : A → **(** S
3 : S → **o** S
4 : S → **)** B
5 : B → **)** B
6 : B → **)**

1°) construisez l'arbre de dérivation dans G de la chaîne : $(^3\ \mathbf{o}^2\ )^4$

2°) construisez un automate fini reconnaissant le langage L(G) :
2.1) en fournissant son graphe
2.2) en indiquant s'il est déterministe ou non
2.3) donnez la séquence d'analyse de la chaîne $(^3\ \mathbf{o}^2\ )^4$

Ex-2 : On donne la Grammaire G_0 de type 3 suivante :

V_N = { ⟨programme⟩ , ⟨instruction⟩ , ⟨affectation⟩ , ⟨terme1⟩, ⟨ terme2⟩, ⟨membre droit⟩ , ⟨oper⟩ }

V_T = { '**a**' , '**b**' , … ,'**z**' , '**fin**' , '**debut**' , '**;**' , '**Lire**', '**Ecrire**' , '+' , '/' , '*****' , '-' , '=' }
*('**a**' , '**b**' , … , '**z**' désigne toutes les lettres de l'alphabet minuscules)*

Axiome : ⟨programme⟩

Règles :
⟨programme⟩ ⟶ **debut** ⟨ instruction⟩
⟨instruction⟩ ⟶ **fin**
⟨instruction⟩ ⟶ **a** ⟨affectation⟩ | **b** ⟨affectation⟩ | ... | **z** ⟨affectation⟩
⟨instruction⟩ ⟶ **Lire** ⟨ terme1⟩ | **Ecrire**⟨ terme1⟩
⟨ terme1⟩ ⟶ **a** ⟨ terme2⟩ | **b** ⟨ terme2⟩ | ... | **z** ⟨terme2⟩
⟨ terme2⟩ ⟶ **;** ⟨instruction⟩
⟨affectation⟩ ⟶ = ⟨membre droit⟩
⟨membre droit⟩ ⟶ **a** ⟨oper⟩ | **b** ⟨oper⟩ | ... | **z** ⟨oper⟩
⟨oper⟩ ⟶ + ⟨ terme1⟩ | ***** ⟨ terme1⟩ | **/** ⟨ terme1⟩ | **-** ⟨ terme1⟩ | ⟨ terme2 ⟩

Cette grammaire G_0 engendre un langage noté $L(G_0)$.

Questions :

1°) combien de règles distinctes possède la grammaire G_0 ?

2°) On donne les deux mots suivants *(les blancs ne sont pas significatifs et ne sont figurés que pour rendre le texte lisible)* :

mot_1 = **debut** **Lire** x ; **Lire** y ; z = x * y ; **Lire** a ; a = a - z ; **Ecrire** a ; **fin**

mot_2 = **debut** **Lire** x ; **Lire** y ; **Lire** a ; a = a - x * y ; **Ecrire** a ; **fin**

En construisant leur arbre de dérivation potentiel dans G_0 , dire si ces deux mots appartiennent ou non au langage de $L(G_0)$.

3°) Construire un automate d'état fini reconnaissant $L(G_0)$ indiquez s'il est déterministe ou non.

4°) Donnez une séquence d'analyse (liste des règles de transition appliquées) de chacun des deux mots de la question 2° et confirmez ainsi votre réponse d'appartenance ou non de ces mots au langage $L(G_0)$.

Ex-3 : Analyse d'une expression arithmétique parenthésée et traduction en expression postfixée, par algorithme d'automate avec pile LIFO. On propose les spécifications suivantes :

Une expression est stockée dans une chaîne de caractères.
- Les expressions sont composées de variables, de chiffres et d'opérateurs
- Les variables ne sont composées que d'une lettre (a,b,c,...z)
- Les chiffres sont : (0, 1, 2, ... ,9)
- Les opérateurs sont : (+, - , * , / , ^ , !)
- Les priorités d'opérateurs sont les suivantes :

+ , - : priorité = 1

/ , * : priorité = 2

^ : priorité = 3

! : priorité = 4

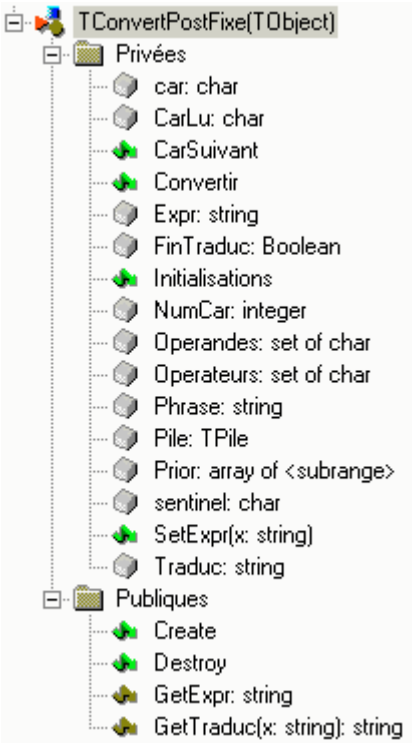
```
Algorithme AutomateParPostFixe
entrée :  phrase //une chaîne contenant l'expression parenthésée
sortie : Traduc //une chaîne contenant l'expression postfixée
local :
 FinAnalyse € booleen,
 CarLu € car,
 sentinelle € car
 Pile //est une pile LIFO contenant des caractères
 EnsdesOpérateurs //ensemble des opérateurs
 EnsdesOpérandes // ensemble des opérandes (variables et chiffres)

 debut
  lire(phrase);
  FinAnalyse <--- Faux ;
  CarLu <--- 1er caractère de phrase ;
  tantque non FinAnalyse faire
   si CarLu € EnsdesOpérandes alors
    concaténer CarLu à Traduc ;
    CarLu suivant ;
   sinon
    si CarLu = ( alors
     Empiler CarLu ;
     CarLu suivant ;
    sinon
     si CarLu € EnsdesOpérateurs alors
      si Sommet de Pile = ( ou Pile est vide alors
       Empiler CarLu ;
       CarLu suivant ;
      sinon //le sommet de pile est un opérateur
       si priorité ( CarLu) > priorité(Sommet de Pile) alors
        Empiler CarLu ;
        CarLu suivant ;
       sinon
        concaténer Sommet de Pile à Traduc
        Dépiler ;
```

```
      fsi
     fsi
    sinon
     si CarLu = ) alors
      si Sommet de Pile appartient  EnsdesOpérateurs  alors
       concaténer Sommet de Pile à Traduc
       Dépiler ;
      sinon //sommet de Pile = (
       Dépiler ;
       CarLu suivant ;
      fsi
     sinon
      si CarLu = sentinelle alors
       si Pile n'est pas vide alors
        concaténer Sommet de Pile à Traduc
        Dépiler ;
       sinon
        FinAnalyse <--- Vrai ;
       fsi
      sinon
       Erreur
      fsi
     fsi
    fsi
   fsi
  fsi
 fsi
ftant
fin // AutomateParPostFixe
```

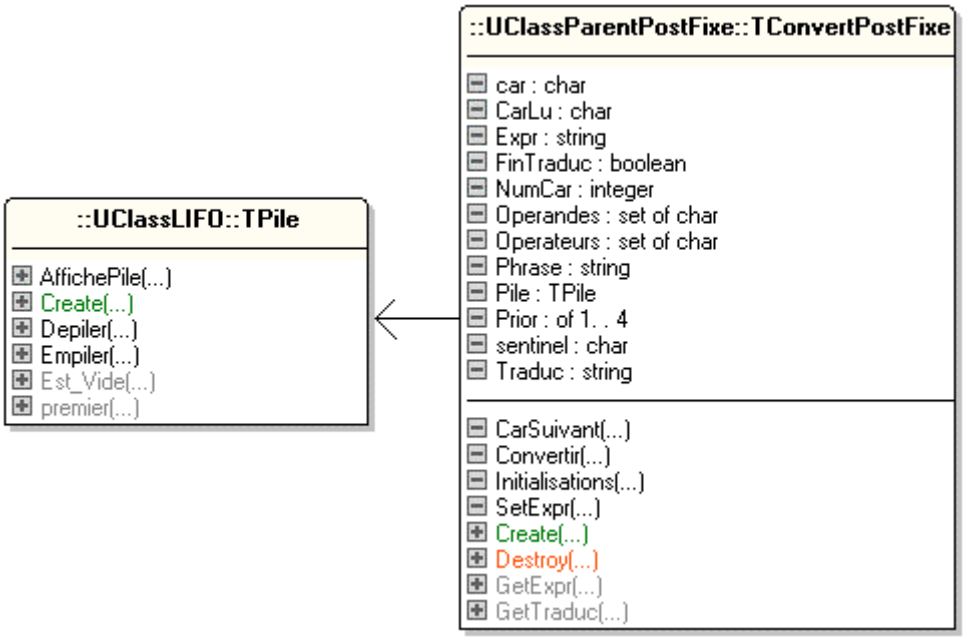
Questions :

1°) Effectuez une trace formelle à figurer dans un tableau de cet algorithme sur l'expression suivante : **(a+b)*c-5**
.

2°) Ci-dessous voici une signature d'une implémentation possible de cet algorithme avec deux classes en Delphi :



On suppose que la classe TPile est déclarée et entièrement implémentée dans la Unit UClassLIFO :



Il vous est demandé de donner le contenu détaillé de la Unit UclassParentPostFixe contenant la classe TconvertPostFixe qui implante l'algorithme précédent.

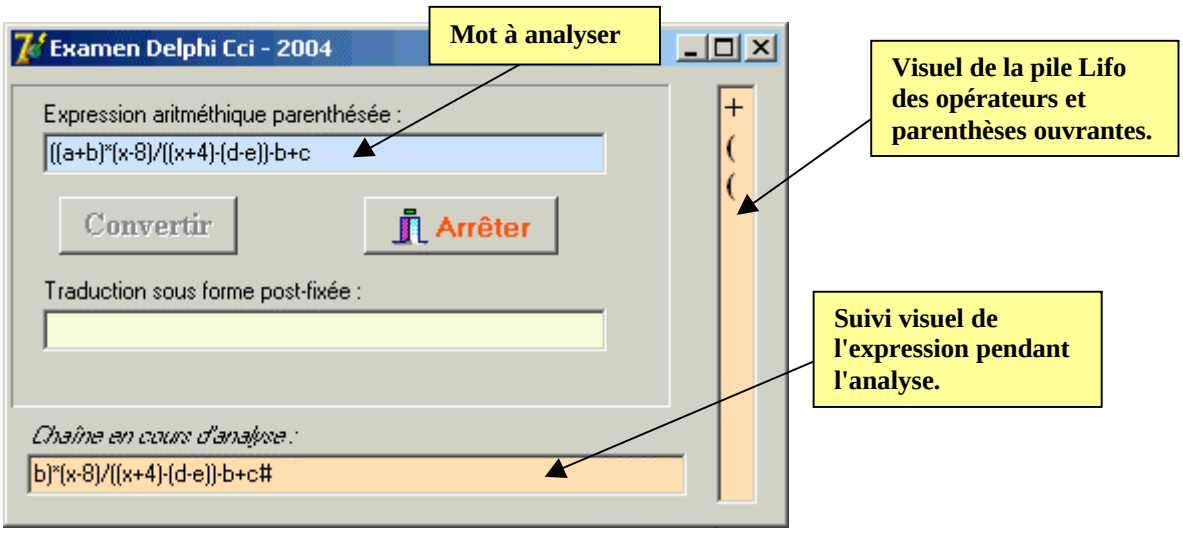
Quelques remarques utiles pour votre implémentation:

Les données:
Expr contient l'expression parenthésée telle qu'elle est rentrée,
Phrase contient l'expression à analyser avec la sentinelle,
Traduc contient l'expression postfixée.

Les méthodes :
GetExpr renvoie l'expression telle qu'elle a été rentrée,
GetTraduc renvoie l'expression une fois traduite en postfixé,
SetExpr prépare les données pour le lancement de la conversion,
Initialisations initialise toutes les données : rempli la table de priorités, les opérateurs, les opérandes…
Convertir : l'algorithme de conversion proprement dit.

3°) On demande de construire une interface d'animation du suivi de l'analyse d'une expression arithmétique.

Cette interface est développée pour tester l'analyseur construit aux questions précédentes. Elle permet d'entrer une expression arithmétique juste, puis de lancer la conversion, alors s'affichent à chaque caractère analysé le contenu de la chaîne en cours d'analyse et le contenu de la pile Lifo des opérateurs et parenthèses ouvrantes.



Travail a effectuer :

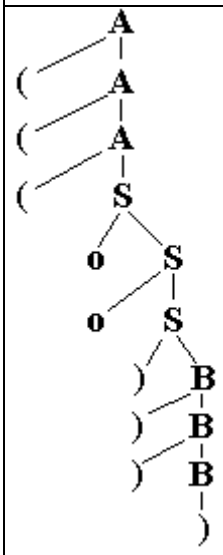
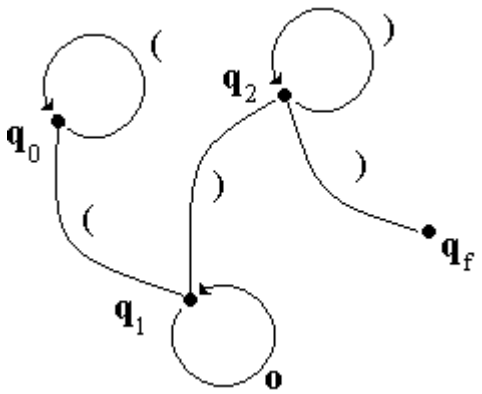
Utiliser dans la classe **TconvertPostFixe** l'agrégation forte d'un objet de classe **TTimer** pour obtenir un programme interactif au sens suivant :

- On peut fermer la fiche en cours d'analyse (à tout moment)
- On peut arrêter à tout moment l'animation de la traduction et la phase de traduction se poursuit alors en mode exécutable sans temporisation.

Rendre un objet de classe **TconvertPostFixe** sensible à un événement que vous dénommerez OnEndTraduction qui se déclenche dès que la traduction de l'expression est terminée, puis dans l'IHM de test interceptez cet événement afin qu'il affiche un message dans une fenêtre de showmessage.

Réponses

Ex-1 :

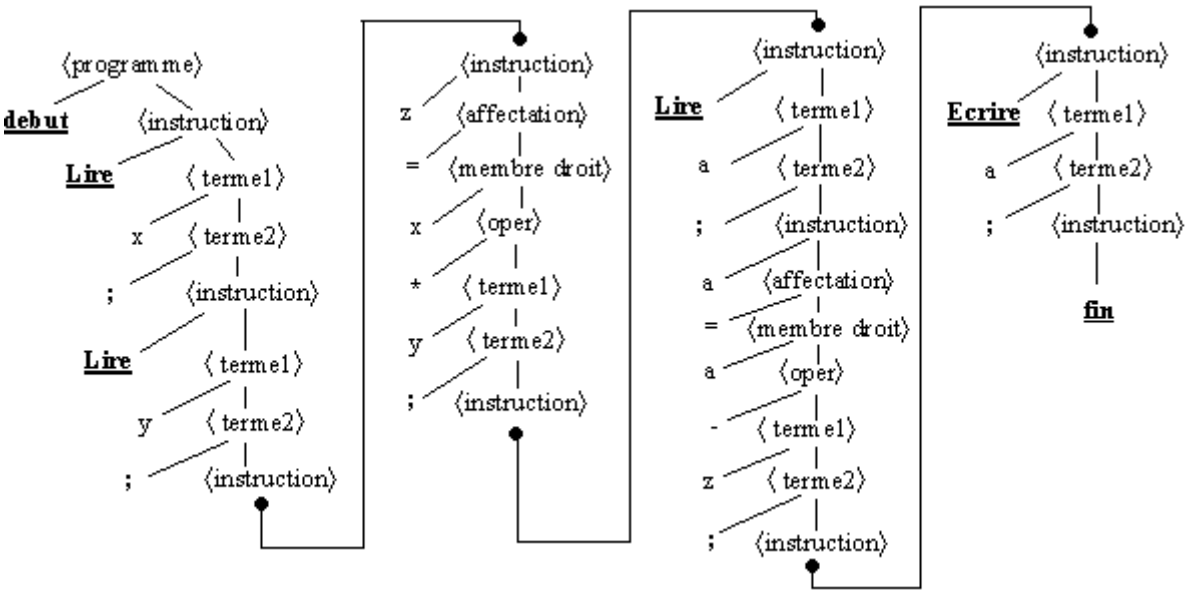
Arbre de dérivation de $({}^{3}\ \mathbf{o}^{2}\ )^{4}$	Automate d'état fini reconnaissant L(G)
	 L'automate est non déterministe. Séquence d'analyse de $({}^{3}\ \mathbf{o}^{2}\ )^{4}$: $(q_0, '(') \rightarrow q_0$ $(q_0, '(') \rightarrow q_0$ $(q_0, '(') \rightarrow q_1$ $(q_1, 'o') \rightarrow q_1$ $(q_1, 'o') \rightarrow q_1$ $(q_1, ')') \rightarrow q_2$ $(q_2, ')') \rightarrow q_2$ $(q_2, ')') \rightarrow q_2$ $(q_2, ')') \rightarrow q_f$

Ex-2 :

1°) <u>Règles</u> :

⟨programme⟩ ⟶ **<u>debut</u>** ⟨ instruction⟩	= 1 règle
⟨instruction⟩ ⟶ **<u>fin</u>**	= 1 règle
⟨instruction⟩ ⟶ **a** ⟨affectation⟩ | **b** ⟨affectation⟩ | ... | **z** ⟨affectation⟩	= 26 règles
⟨instruction⟩ ⟶ **<u>Lire</u>** ⟨ terme1⟩ | **<u>Ecrire</u>**⟨ terme1⟩	= 2 règles
⟨ terme1⟩ ⟶ **a** ⟨ terme2⟩ | **b** ⟨ terme2⟩ | ... | **z** ⟨terme2⟩	= 26 règles
⟨ terme2⟩ ⟶ **;** ⟨instruction⟩	= 1 règle
⟨affectation⟩ ⟶ = ⟨membre droit⟩	= 1 règle
⟨membre droit⟩ ⟶ **a** ⟨oper⟩ | **b** ⟨oper⟩ | ... | **z** ⟨oper⟩	= 26 règles
⟨oper⟩ ⟶ + ⟨ terme1⟩ | ***** ⟨ terme1⟩ | / ⟨ terme1⟩ | - ⟨ terme1⟩ | ⟨ terme2 ⟩	= 5 règles
	= 26 règles
Total	= 115 règles

2°) Arbre de dérivation de mot_1 = **<u>debut</u>** **<u>Lire</u>** x ; **<u>Lire</u>** y ; z = x * y ; **<u>Lire</u>** a ; a = a - z ; **<u>Ecrire</u>** a ; **<u>fin</u>**



celui de mot_2 ne peut être construit en entier car l'instruction a = a - x * y ; n'est pas dérivable dans la grammaire.
Donc mot_1 appartient à $L(G_0)$, mot n'appartient pas à $L(G_0)$.

3°)

AEF donné par ses règles		
($\mathbf{q}_0$, **<u>debut</u>**) → $\mathbf{q}_1$ ($\mathbf{q}_1$, **<u>fin</u>**) → $\mathbf{q}_f$ ($\mathbf{q}_1$, *Lettres*) → $\mathbf{q}_2$ ($\mathbf{q}_2$, =) → $\mathbf{q}_5$ ($\mathbf{q}_5$, *Lettres*) → $\mathbf{q}_6$ ($\mathbf{q}_4$, ;) → $\mathbf{q}_1$	($\mathbf{q}_6$, *Operat*) → $\mathbf{q}_3$ ($\mathbf{q}_6$, ;) → $\mathbf{q}_1$ ($\mathbf{q}_1$, **<u>Lire</u>**) → $\mathbf{q}_3$ ($\mathbf{q}_1$, **<u>Ecrire</u>**) → $\mathbf{q}_3$ ($\mathbf{q}_3$, *Lettres*) → $\mathbf{q}_4$	*Lettres = { a, b, … , z }* *Operat = { +,- , /,* }*

Reconnaissance de **mot_1**				
($\mathbf{q}_0$, **<u>debut</u>**) → $\mathbf{q}_1$ ($\mathbf{q}_1$, **<u>Lire</u>**) → $\mathbf{q}_3$ ($\mathbf{q}_3$, x) → $\mathbf{q}_4$ ($\mathbf{q}_4$, ;) → $\mathbf{q}_1$ ($\mathbf{q}_1$, **<u>Lire</u>**) → $\mathbf{q}_3$	($\mathbf{q}_3$, y) → $\mathbf{q}_4$ ($\mathbf{q}_4$, ;) → $\mathbf{q}_1$ ($\mathbf{q}_1$, z) → $\mathbf{q}_2$ ($\mathbf{q}_2$, =) → $\mathbf{q}_5$ ($\mathbf{q}_5$, x) → $\mathbf{q}_6$	($\mathbf{q}_6$, *) → $\mathbf{q}_3$ ($\mathbf{q}_3$, y) → $\mathbf{q}_4$ ($\mathbf{q}_4$, ;) → $\mathbf{q}_1$ ($\mathbf{q}_1$, **<u>Lire</u>**) → $\mathbf{q}_3$ ($\mathbf{q}_3$, a) → $\mathbf{q}_4$	($\mathbf{q}_4$, ;) → $\mathbf{q}_1$ ($\mathbf{q}_1$, a) → $\mathbf{q}_2$ ($\mathbf{q}_2$, =) → $\mathbf{q}_5$ ($\mathbf{q}_5$, a) → $\mathbf{q}_6$ ($\mathbf{q}_6$, -) → $\mathbf{q}_3$	($\mathbf{q}_3$, z) → $\mathbf{q}_4$ ($\mathbf{q}_4$, ;) → $\mathbf{q}_1$ ($\mathbf{q}_1$, **<u>Ecrire</u>**) → $\mathbf{q}_3$ ($\mathbf{q}_3$, y) → $\mathbf{q}_4$ ($\mathbf{q}_4$, ;) → $\mathbf{q}_1$ ($\mathbf{q}_1$, **<u>fin</u>**) → $\mathbf{q}_f$

Tentative de reconnaissance de **mot$_2$**			
(q_0 , **debut**) → q_1 (q_1 , **Lire**) → q_3 (q_3 , x) → q_4 (q_4 , ;) → q_1 (q_1 , **Lire**) → q_3	(q_3 , y) → q_4 (q_4 , ;) → q_1 (q_1 , **Lire**) → q_3 (q_3 , a) → q_4 (q_4 , ;) → q_1	(q_1 , a) → q_2 (q_2 , =) → q_5 (q_5 , a) → q_6 (q_6 , -) → q_3 (q_3 , x) → q_4	(q_4 , *) → **??** aucune règle de la forme (q_4 , *) → … Donc mot$_2$ n'est pas reconnu.

Remarquons que mot$_2$ est partiellement conforme à la syntaxe jusqu'à :
debut **Lire** x ; **Lire** y ; **Lire** a ; a = a - x ← le reste n'est pas correct

Ex-3 : Solution pratique de "expression arithmétique parenthésée"

Questions 2°)

```
unit UClassLIFO;
{une implantation en Delphi du TAD Pile LIFO
à partir de la classe tlist     }

interface

 uses classes;

 type
  TPile = class(TList)
 public
  constructor Create;
  function  Est_Vide : boolean;
  procedure Empiler (elt: char);
  procedure Depiler (var elt: char);
  function  premier : char;
  procedure AffichePile;
end;


implementation

 constructor TPile.Create;
 begin
  inherited Create;
 end;

 function TPile.Est_Vide: boolean;
 begin
  if count = 0 then
   result := true
  else
   result := false
 end;

 procedure TPile.Empiler (elt: char);
 begin
  self.Capacity := Count;
  Add(TObject(elt))
 end;
```

```
procedure TPile.Depiler (var elt: char);
begin
 if not Est_Vide then
 begin
  elt:=char(Last);
  self.Delete(count-1);
  self.Pack;
  self.Capacity := Count;
 end
else
  writeln(' désolé pile vide !');
end;

function TPile.premier : char;
begin
 if not est_vide then
  result:=char(last)
 else
  result:='@';
end;

procedure TPile.AffichePile;
 var i:integer;
 begin
  if not Est_Vide then
  begin
   writeln('Pile contenant : ',count,' éléments.');
   for i:=0 to count-1 do
    write(char(Items[i]));
   writeln
  end
  else
   writeln('pile vide.');
 end;

end.

unit UClassParentPostFixe;
{implantation en Delphi d'un convertisseur d'expression parenthésée en postfixé
avec pile lifo       }

interface

 uses UClassLIFO;

 type
 { Grammaire :
    opérandes possibles: a,b,...,z,0,1,...,9
    opérateurs possibles: +,-,/,*,^,!
       +,- : prior = 1

       /,* : prior = 2

        ^ : prior = 3

        ! : prior = 4
 }
 TConvertPostFixe = class
 private
  Pile:TPile;
```

Ecriture d'informations sur la console.

```
  Phrase,Expr,Traduc:string;
  CarLu,car,sentinel:char;
  NumCar:integer;
  Operandes,Operateurs:set of char;
  Prior:array[char] of 1..4;
  FinTraduc:Boolean;
  procedure Initialisations;
  procedure CarSuivant;
  procedure SetExpr(x:string);
  procedure Convertir;
 public
  constructor Create;
  destructor Destroy;
  function GetExpr:string;
  function GetTraduc(x:string):string;
end;

implementation
uses Dialogs;

 // Private --->
 procedure TConvertPostFixe.Initialisations;
 begin
  Prior['+']:=1;
  Prior['-']:=1;
  Prior['/']:=2;
  Prior['*']:=2;
  Prior['^']:=3;
  Prior['!']:=4;
  Operandes:=['a'..'z']+['0'..'9'];
  Operateurs:=['+','-','/','*','^','!'];
  NumCar:=0;
  Traduc:='';
  FinTraduc:=false;
 end;

 procedure TConvertPostFixe.CarSuivant;
 begin
  NumCar:=NumCar+1;
  CarLu:=Phrase[NumCar]
 end;

 procedure TConvertPostFixe.SetExpr(x:string);
 begin
  Traduc:='';
  Expr:=x;
  Phrase:=Expr+sentinel;
 end;

 procedure TConvertPostFixe.Convertir;
 begin
  CarSuivant;
  while not FinTraduc do
  begin
   if CarLu in Operandes then
   begin
    Traduc:=Traduc+CarLu;
    CarSuivant;
   end
   else
```

```
  if carlu='(' then
  begin
   Pile.Empiler(CarLu);
   CarSuivant;
  end
  else
   if carlu in operateurs then
   begin
    if (Pile.premier='(')or(Pile.Est_Vide) then
    begin
     Pile.Empiler(CarLu);
     CarSuivant;
    end
    else {sommet de pile est un opérateur}
     if Prior[CarLu] > Prior[Pile.premier] then
     begin
      Pile.Empiler(CarLu);
      CarSuivant;
     end
     else
     begin
      Traduc:=Traduc+Pile.premier;
      Pile.Depiler(car);
     end
    end
    else
     if carlu=')' then
     begin
      if Pile.premier in Operateurs then
      begin
       Traduc:=Traduc+Pile.premier;
       Pile.Depiler(car);
      end
      else {sommet de pile = '('}
      begin
       Pile.Depiler(car);
       CarSuivant;
      end
     end
     else
      if carlu = sentinel then
      begin
       if not Pile.Est_Vide then
       begin
        Traduc:=Traduc+Pile.premier;
        Pile.Depiler(car);
       end
       else
        fintraduc:=true;
      end
 end;
end;

// Public --->
constructor TConvertPostFixe.Create;
begin
 sentinel:='#';
 Pile:=TPile.Create;
 Initialisations;
end;
```

```
destructor TConvertPostFixe.Destroy;
begin
 Pile.Free;
 Pile:=nil;
 inherited;
end;

function TConvertPostFixe.GetExpr:string;
begin
 result:= Expr
end;

function TConvertPostFixe.GetTraduc(x:string):string;
begin
 SetExpr(x);
 Convertir;
 if length(traduc)=0 then
  showmessage('expression vide');
 result:=Traduc
end;

end.
```

Version IHM et animation

```
unit UClassLIFO;
{une implantation en Delphi du TAD Pile LIFO
à partir de la classe tlist     }

interface

 uses classes;

 type
  TPile = class(TList)
 public
  constructor Create;
  function  Est_Vide : boolean;
  procedure Empiler (elt: char);
  procedure Depiler (var elt: char);
  function  premier : char;
 // procedure AffichePile;
end;


implementation

 constructor TPile.Create;
 begin
  inherited Create;
 end;

 function TPile.Est_Vide: boolean;
 begin
  if count = 0 then
   result := true
  else
```

```
  result := false
end;

procedure TPile.Empiler (elt: char);
begin
 self.Capacity := Count;
 Add(TObject(elt))
end;

procedure TPile.Depiler (var elt: char);
begin
 if not Est_Vide then
 begin
  elt:=char(Last);
  self.Delete(count-1);
  self.Pack;
  self.Capacity := Count;
 end
// else
//  writeln(' désolé pile vide !');
end;

function TPile.premier : char;
begin
 if not est_vide then
  result:=char(last)
 else
  result:='@';
end;
{
procedure TPile.AffichePile;
var i:integer;
begin
 if not Est_Vide then
 begin
  writeln('Pile contenant : ',count,' éléments.');
  for i:=0 to count-1 do
   write(char(Items[i]));
  writeln
 end
 else
  writeln('pile vide.');
end;
}
end.
```

On enlève dans la classe précédente tout relation avec une écriture sur la console.

```
unit UClassParentPostFixe;
{implantation en Delphi d'un convertisseur d'expression parenthésée correcte en postfixé avec pile lifo      }

interface

 uses UClassLIFO,Classes,StdCtrls,SysUtils,Forms,ExtCtrls;

 type
 {
    opérandes possibles: a,b,...,z,0,1,...,9
    opérateurs possibles: +,-,/,*,^,!
       +,- : prior = 1
       /,* : prior = 2
```

```
      ^ : prior = 3
      ! : prior = 4
}
TConvertPostFixe = class(TComponent)
 private
  FonEndTraduction:TNotifyEvent;
  dureefinie:boolean;
  Farret:boolean;
  FTimer:TTimer; //agrégation forte
  Flifo:TListBox; // référence vers un TListBox : agrégation faible
  PhraseEncours:Tedit; // référence vers un TEdit : agrégation faible
  Pile:TPile;
  Phrase,Expr,Traduc:string;
  CarLu,car,sentinel:char;
  NumCar:integer;
  Operandes,Operateurs:set of char;
  Prior:array[char] of 1..4;
  FinTraduc:Boolean;
  procedure Initialisations;
  procedure CarSuivant;
  procedure SetExpr(x:string);
  procedure Convertir;
  procedure Temporiser;
  function Gettempo:integer;
  procedure Settempo(x:integer);
  procedure OnTimerFTimer(Sender:TObject);
  procedure Setarret(x:boolean);
  function Getarret:boolean;
 public
  constructor Create(lifo:TListBox;listc:TEdit);reintroduce;overload;
  constructor Create(lifo:TListBox;listc:TEdit;temps:integer);reintroduce;overload;
  destructor Destroy;override;
  function GetExpr:string;
  function GetTraduc(x:string):string;
 published
  property tempo:integer read Gettempo write Settempo;
  property arret:boolean read Getarret write Setarret;
  property OnEndTraduction:TNotifyEvent read FonEndTraduction write FonEndTraduction;
end;

implementation
uses Dialogs;

 // Private --->
 procedure TConvertPostFixe.Initialisations;
 begin
  Prior['+']:=1;
  Prior['-']:=1;
  Prior['/']:=2;
  Prior['*']:=2;
  Prior['^']:=3;
  Prior['!']:=4;
  Operandes:=['a'..'z']+['0'..'9'];
  Operateurs:=['+','-','/','*','^','!'];
  NumCar:=0;
  Traduc:='';
  FinTraduc:=false;
  Farret:=false;
 end;
```

```
procedure TConvertPostFixe.CarSuivant;
begin
 NumCar:=NumCar+1;
 CarLu:=Phrase[NumCar];
 PhraseEncours.Text:=copy(Phrase,NumCar,length(Phrase));
 if not Farret then
  Temporiser; // temporisation pour animation
end;

procedure TConvertPostFixe.SetExpr(x:string);
begin
 Traduc:='';
 Expr:=x;
 Phrase:=Expr+sentinel;
end;

procedure TConvertPostFixe.Convertir;
begin
 CarSuivant;
 while not FinTraduc do
 begin
  if CarLu in Operandes then
  begin
   Traduc:=Traduc+CarLu;
   CarSuivant;
  end
  else
   if carlu='(' then
   begin
    Pile.Empiler(CarLu);
    CarSuivant;
   end
   else
    if carlu in operateurs then
    begin
     if (Pile.premier='(')or(Pile.Est_Vide) then
     begin
      Pile.Empiler(CarLu);
      CarSuivant;
     end
     else {sommet de pile est un opérateur}
      if Prior[CarLu] > Prior[Pile.premier] then
      begin
       Pile.Empiler(CarLu);
       CarSuivant;
      end
      else
      begin
       Traduc:=Traduc+Pile.premier;
       Pile.Depiler(car);
      end
    end
    else
     if carlu=')' then
     begin
      if Pile.premier in Operateurs then
      begin
       Traduc:=Traduc+Pile.premier;
       Pile.Depiler(car);
      end
```

```
      else {sommet de pile = '('}
      begin
       Pile.Depiler(car);
       CarSuivant;
      end
     end
     else
      if carlu = sentinel then
      begin
       if not Pile.Est_Vide then
       begin
        Traduc:=Traduc+Pile.premier;
        Pile.Depiler(car);
       end
       else
       begin
        FinTraduc:=true;
        if Assigned(FOnEndTraduction)then
         FOnEndTraduction(self)
       end
      end
 end;
end;

// Public --->
constructor TConvertPostFixe.Create(lifo:TListBox;listc:TEdit);
begin
 sentinel:='#';
 Flifo:=lifo;
 dureefinie:=true;
 PhraseEncours:= listc;
 Pile:=TPile.Create(lifo);
 FTimer:=TTimer.Create(self);
 FTimer.Interval:=1000;
 FTimer.Enabled:=false;
 FTimer.OnTimer:=OnTimerFTimer;
 Initialisations;
end;

constructor TConvertPostFixe.Create(lifo:TListBox;listc:TEdit;temps:integer);
begin
 sentinel:='#';
 Flifo:=lifo;
 dureefinie:=true;
 PhraseEncours:= listc;
 Pile:=TPile.Create(lifo);
 FTimer:=TTimer.Create(self);
 FTimer.Interval:=temps;
 FTimer.Enabled:=false;
 FTimer.OnTimer:=OnTimerFTimer;
 Initialisations;
end;

destructor TConvertPostFixe.Destroy;
begin
 Pile.Free;
 Pile:=nil;
 inherited;
end;
```

OnEndTraduction est donc déclenché dès que la traduction de l'expression est terminée.

```
function TConvertPostFixe.GetExpr:string;
begin
 result:= Expr
end;

function TConvertPostFixe.GetTraduc(x:string):string;
begin
 Initialisations;
 SetExpr(x);
 Convertir;
 if length(traduc)=0 then
  showmessage('expression vide');
 result:=Traduc
end;

procedure TConvertPostFixe.Temporiser;
begin
 dureefinie:=false;
 FTimer.Enabled:=true;
 repeat
  Application.ProcessMessages
 Until  (ftimer.enabled=false)or(application.terminated);
 Flifo.Repaint;
 PhraseEncours.Repaint;
end;

function TConvertPostFixe.Gettempo: integer;
begin
 result:=FTimer.Interval
end;

procedure TConvertPostFixe.Settempo(x: integer);
begin
 if x>0 then
  FTimer.Interval:=x
 else
  FTimer.Interval:=100
end;

procedure TConvertPostFixe.OnTimerFTimer(Sender: TObject);
begin
 if dureefinie=true then
  FTimer.Enabled:=false
 else
  dureefinie:=true
end;

function TConvertPostFixe.Getarret: boolean;
begin
 result:=Farret;
end;

procedure TConvertPostFixe.Setarret(x: boolean);
begin
 Farret:=x;
 FTimer.Enabled:=not x;
end;

end.
```

Boucle infinie arrêtée par le Timer.

Chapitre 7 : Communication homme-machine

7.1. Les interfaces de communication logiciel/utilisateur

- Objets d'E/S
- temps d'attente
- pilotage
- enchaînement
- résistance aux erreurs

7.2. Grammaire pour analyser des phrases

- Un retour sur les grammaires
- Grammaire LL(1) pour analyse déterministe
- Analyser des phrases

7.3. Interface et pilotage en mini-français

- Un peu plus loin avec l'interaction et le pilotage
- Une méthode de construction

7.4. Projet d'IHM : enquête fumeurs

- Construction d'une borne interactive
- Mode saisie et plans d'actions
- Reste du logiciel

7.5. Utilisation des bases de données

- Introduction et généralités
- Le modèle de données relationnel
- Principes fondamentaux d'une algèbre relationnelle
- SQL et algèbre relationnelle
- Exemples de communication de Delphi avec une BD

Chapitre 7.1 interfaces de communication logiciel / utilisateur

Plan du chapitre:

Introduction

Interface homme-machine les concepts :

- **Les objets d'entrée-sortie**
- **Les temps d'attente**
- **Le pilotage de l'utilisateur**
- **Les types d'interaction**
- **L'enchaînement des opérations**
- **La résistance aux erreurs**

Introduction

Nous énumérons quelques principes utiles à l'élaboration d'une interface associée étroitement à la programmation événementielle. Le lecteur qui connaît le sujet peut passer au chapitre suivant.
Notre point de vue reste celui du pédagogue et non pas du spécialiste en ergonomie ou en psychologie cognitive qui sont deux éléments essentiels dans la conception d'une interface homme-machine (**IHM**). Nous nous efforcerons d'utiliser les principes généraux des **IHM** en les mettant à la portée d'un débutant avec un double objectif :

- Faire écrire des programmes interactifs. Ce qui signifie que le programme doit communiquer avec l'utilisateur qui reste l'acteur privilégié de la communication. Les programmes sont Windows-like et nous nous servons du RAD visuel Delphi pour les développer. Une partie de la spécification des programmes s'effectue avec des objets graphiques représentant des classes(programmation objet visuelle).

- Le programmeur peut découpler pendant la conception la programmation de son interface de la programmation des tâches internes de son logiciel (pour nous généralement ce sont des algorithmes ou des scénarios objets).

Nous nous efforçons aussi de ne proposer que des outils pratiques qui sont à notre portée et utilisables rapidement avec un système de développement RAD visuel comme Delphi.

Interface homme-machine, les concepts

Les spécialistes en ergonomie conceptualisent une IHM en six concepts :

- les objets d'entrée-sortie,
- les temps d'attente (temps de réponse aux sollicitations),
- le pilotage de l'utilisateur dans l'interface,
- les types d'interaction (langage,etc..) ,
- l'enchaînement des opérations,
- la résistance aux erreurs (ou robustesse qui est la qualité qu'un logiciel à fonctionner même dans des conditions anormales).

Un principe général provenant des psycho-linguistes guide notre programmeur dans la complexité informationnelle : la mémoire rapide d'un humain ne peut être sollicitée que par un nombre limité de concepts différents en même temps (nombre compris entre sept et neuf). Développons un peu plus chacun des six concepts composants une interface.

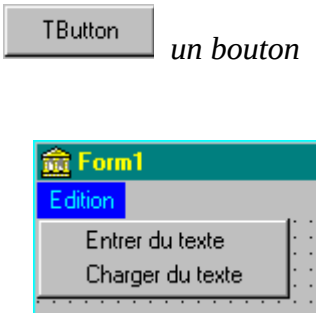
Les objets d'entrée-sortie

Concept

Une IHM présente à l'utilisateur un éventail d'informations qui sont de deux ordres : des commandes entraînant des actions internes sur l'IHM et des données présentées totalement ou partiellement selon l'état de l'IHM.

Les commandes participent à la " saisie de l'intention d'action" de l'utilisateur, elles sont matérialisées dans le dialogue par des ***objets d'entrée*** de l'information (boîtes de saisie, boutons, menus etc...).

Voici avec un RAD visuel comme Delphi, des objets visuels associés aux objets d'entrée de l'information , ils sont très proches visuellement des objets que l'on trouve dans d'autres RAD visuels, car en fait ce sont des surcouches logiciels de contrôles de base du système d'exploitation (qui est lui-même fenêtré et se présente sous forme d'une IHM dénommée bureau électronique).



un bouton

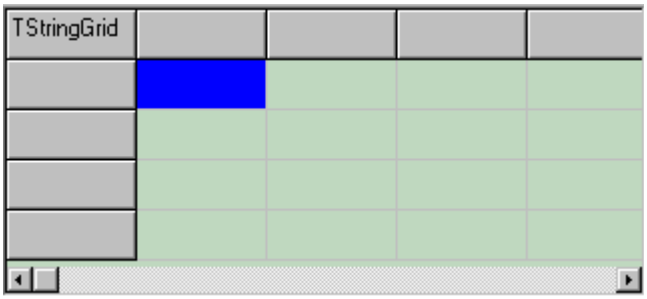
Un menu

une boîte de saisie mono-ligne

une boîte de saisie multi-ligne

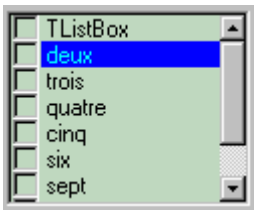
Les données sont présentées à un instant précis du dialogue à travers des ***objets de sortie*** de l'information (boîte d'édition monoligne, multiligne, tableaux, graphiques, images, sons etc...).

Ci-dessous quelques objets visuels associés à des objets de sortie de l'information :

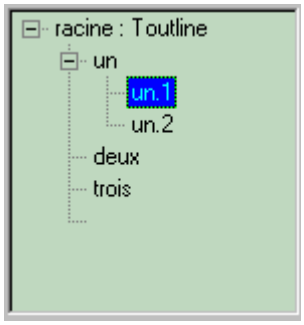


un TstringGrid (tableau)

un Timage(image jpg, png, bm ,ico,...)



un TlistBox (liste)



un Touline (arbre)

Les temps d'attente

Concept

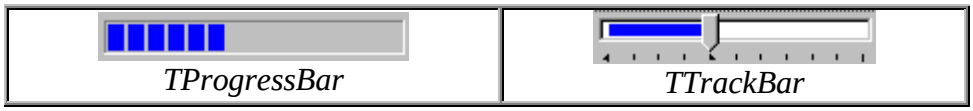
Sur cette question, une approche psychologique est la seule réponse possible, car l'impression d'attente ne dépend que de celui qui attend selon son degré de patience. Toutefois, puisque

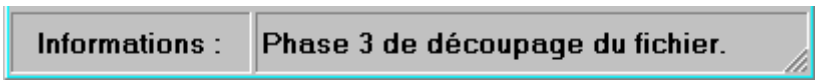
nous avons parlé de la mémoire à court terme (mémoire rapide), nous pouvons nous baser sur les temps de persistance généralement admis (environ 5 secondes).

Nous considérerons qu'en première approximation, si le délai d'attente est :

- inférieur à environ une seconde la réponse est quasi-instantanée,
- compris entre une seconde et cinq secondes il y a attente, toutefois la mémoire rapide de l'utilisateur contient encore la finalité de l'opération en cours.
- lorsque l'on dépasse la capacité de mémorisation rapide de l'utilisateur alors il faut soutenir l'attention de l'utilisateur en lui envoyant des informations sur le déroulement de l'opération en cours (on peut utiliser pour cela par exemple des barres de défilement, des jauges, des boîtes de dialogue, etc...)

Exemples de quelques classes d'objets visuels de gestion du délai d'attente :





TStatusBar (avec deux panneaux ici)

Le pilotage de l'utilisateur

Concept

Nous ne cherchons pas à explorer les différentes méthodes utilisables pour piloter un utilisateur dans sa navigation dans une interface. Nous adoptons plutôt la position du concepteur de logiciel qui admet que le futur utilisateur ne se servira de son logiciel que d'une façon épisodique. Il n'est donc pas question de demander à l'utilisateur de connaître en permanence toutes les fonctionnalités du logiciel.

En outre, il ne faut pas non plus submerger l'utilisateur de conseils de guides et d'aides à profusion, car ils risqueraient de le détourner de la finalité du logiciel. Nous préférons adopter une ligne moyenne qui consiste à fournir de petites aides rapides contextuelles (au moment où l'utilisateur en a besoin) et une aide en ligne générale qu'il pourra consulter s'il le souhaite. Ce qui revient à dire que l'on accepte deux niveaux de navigation dans un logiciel :

- le niveau de surface permettant de réagir aux principales situations,
- le niveau approfondi qui permet l'utilisation plus complète du logiciel.

Il faut admettre que le niveau de surface est celui qui restera le plus employé (l'exemple d'un logiciel de traitement de texte courant du commerce montre qu'au maximum 30% des fonctionnalités du produit sont utilisées par plus de 90% des utilisateurs).

Pour permettre un pilotage plus efficace on peut établir à l'avance un graphe d'actions possibles du futur utilisateur (nous nous servirons du graphe événementiel) et ensuite diriger l'utilisateur dans ce graphe en matérialisant (masquage ou affichage) les actions qui sont

réalisables. En application des notions acquises dans les chapitres précédents, nous utiliserons un pilotage dirigé par la syntaxe comme exemple.

Les types d'interaction

Concept

Le tout premier genre d'interaction entre l'utilisateur et un logiciel est apparu sur les premiers systèmes d'exploitation sous la forme d'un langage de commande. L'utilisateur dispose d'une famille de commandes qu'il est censé connaître, le logiciel étant doté d'une interface interne (l'interpréteur de cette famille de commandes). Dès que l'utilisateur tape textuellement une commande (exemple MS-DOS " **dir c: /w** "), le système l'interprète (dans l'exemple : lister en prenant toutes les colonnes d'écran, les bibliothèques et les fichiers du disque C).

Nous adoptons comme mode d'interaction entre un utilisateur et un logiciel, une extension plus moderne de ce genre de dialogue, en y ajoutant, en privilégiant, la notion d'objets visuels permettant d'effectuer des commandes par actions et non plus seulement par syntaxe textuelle pure.
Nous construisons donc une interface tout d'abord ***essentiellement*** à partir des interactions événementielles, puis lorsque cela est utile ou nécessaire, nous pouvons ajouter un interpréteur de langage (nous pouvons par exemple utiliser des automates d'états finis pour la reconnaissance).

L'enchaînement des opérations

Concept

Nous savons que nous travaillons sur des machines de Von Neumann, donc séquentielles, les opérations internes s'effectuant selon un ordre unique sur lequel l'utilisateur n'a aucune prise. L'utilisateur est censé pouvoir agir d'une manière " aléatoire ". Afin de simuler une certaine liberté d'action de l'utilisateur nous lui ferons parcourir un graphe événementiel prévu par le programmeur. Il y a donc contradiction entre la rigidité séquentielle imposée par la machine et la liberté d'action que l'on souhaite accorder à l'utilisateur. Ce problème est déjà présent dans un système d'exploitation et il relève de la notion de gestion des interruptions.

Nous pouvons trouver un compromis raisonnable dans le fait de découper les tâches internes en tâches séquentielles minimales ininterruptibles et en tâches interruptibles.

Les interruptions consisteront en actions potentielles de l'utilisateur sur la tâche en cours afin de :

- interrompre le travail en cours,
- quitter définitivement le logiciel,
- interroger un objet de sortie,
- lancer une commande exploratoire ...

Il faut donc qu'existe dans le système de développement du logiciel, un mécanisme qui permette de " *demander la main au système* " sans arrêter ni bloquer le reste de l'interface, ceci pendant le déroulement d'une action répétitive et longue. Lorsque l'interface a la main, l'utilisateur peut alors interrompre, quitter, interroger...

Ce mécanisme est disponible dans les RAD visuels pédagogiques (Delphi,Visual Basic, Visual C#), nous verrons comment l'implanter. Terminons ce tour d'horizon, par le dernier concept de base d'une interface : sa capacité à absorber certains dysfonctionnements.

La résistance aux erreurs

Concept

Il faut en effet employer une méthode de programmation défensive afin de protéger le logiciel contre des erreurs comme par exemple des erreurs de manipulation de la part de l'utilisateur. Nous utilisons plusieurs outils qui concourent à la robustesse de notre logiciel. La protection est donc située à plusieurs niveaux.

1°) Une protection est apportée par le graphe événementiel qui n'autorise que certaines actions (activation-désactivation), matérialisé par un objet tel qu'un menu :

l'item " taper un fichier " n'est pas activable

l'item " Charger un fichier " est activable

2°) Une protection est apportée par le filtrage des données (on peut utiliser par exemple des logiciels d'automates de reconnaissance de la syntaxe des données).

3°) Un autre niveau de protection est apporté par les composants visuels utilisés qui sont sécurisés dans le RAD à l'origine. Par exemple la méthode LoadFromfile de Delphi qui permet le chargement d'un fichier dans un composant réagit d'une manière sécuritaire (c'est à dire rien ne se produit)lorsqu'on lui fournit un chemin erroné ou que le fichier n'existe pas.

4°) Un niveau de robustesse est apporté en Delphi par une utilisation des exceptions (semblable à Ada ou à C++) autorisant le détournement du code afin de traiter une situation interne anormale (dépassement de capacité d'un calcul, transtypage de données non conforme etc...). Le programmeur peut donc prévoir les incidents possibles et construire des gestionnaires d'exceptions.

Chapitre 7.2 grammaire pour analyser des phrases

Plan du chapitre:

1. Un retour sur les grammaires

1.1 Dérivation à droite et à gauche
1.2 Ensembles First et Init
1.3 Ensemble Follow
1.4 Calcul des Follows à partir des First
1.5 Calcul pour une grammaire arithmétique
1.6 Calcul pour une grammaire du mini-français

2. Grammaire LL(1) pour analyse déterministe

2.1 Une condition pratique d'analysabilité LL(1)
2.2 Méthode pratique de vérification

3. Analyser des phrases

2.3 Une grammaire GF2 de type LL(1) du mini-français
2.4 Schémas d'algorithmes associés à G_{F2}
2.5 Code Delphi associé aux blocs dans G_{F2}

Bien qu'il ne soit pas dans les objectifs de cet ouvrage de systématiser les méthodes de reconnaissance, nous abordons le sujet sur des cas et des exemples particuliers. L'attitude de systématisation méthodologique et pratique adoptée devrait fournir matière à réflexion et profiter à l'étudiant, nous nous servirons de ces outils pour améliorer ses IHM en particulier dans l'analyse des interactions avec l'utilisateur.

1. Un retour sur les grammaires

L'expérience a montré qu'un des cas particuliers abordables en initiation est celui où une C-grammaire G possède la propriété d'être analysable par analyse LL(1). Ces analyseurs sont plus simples à construire, et surtout il est possible de systématiser leur construction. Il apparaît que beaucoup de grammaires sont LL(1) et qu'un très grand nombre d'exemples du niveau étudiant débutant peuvent être décrits par une grammaire LL(1). C'est pourquoi nous fournirons plus loin un exemple de génération manuelle d'un petit analyseur de mots du genre LL(1) à partir d'un TAD, ainsi qu'une définition d'une grammaire LL(1).

1.1 Dérivation à droite et à gauche

Nous dirons par définition que x se ***dérive le plus à gauche*** en y et l'on écrira :

$x \overset{\Rightarrow}{_{lpg}} y$ *si et seulement si :*	$\exists\ \alpha \in (V_N \cup V_T)^*$ $\exists\ r_i$: A $\rightarrow$ avec A $\in V_N$ si x = Aα alors y = βα

Nous dirons par définition que x se ***dérive le plus à droite*** en y et l'on écrira :

$x \overset{\Rightarrow}{_{lpd}} y$ *si et seulement si :*	$\exists\ \alpha \in (V_N \cup V_T)^*$ $\exists\ r_i$: A $\rightarrow$ avec A $\in V_N$ si x = α A alors y = αβ

Comme pour la dérivation, on définit la fermeture transitive de chacune de ces deux relations binaires. Nous les notons :

$x \overset{\Rightarrow}{_{lpd}}{}^{*}\ y$ **et** $x \overset{\Rightarrow}{_{lpg}}{}^{*}\ y$.

1.2 Ensembles First et Init

Nous allons définir quelques ensembles de symboles utiles à une reconnaissance des mots dans une grammaire G, G = (V_N,V_T,S,R).

Notations : Les ensembles First et leurs propriétés

Soient $(x,y) \in (V_N \cup V_T)^{*2}$, $A \in V_N$ (on suppose que A possède dans le cas général plusieurs expansions $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, le symbole S est l'axiome de la grammaire G. Nous notons :

$$\textbf{First(x)} = \{\mathbf{a} \in V_T \ / \ x \overset{\Rightarrow}{\scriptstyle lpg} * \mathbf{a}y\}$$

Autrement dit pour l'élément $a \in$ First(x) nous avons :

a est un symbole du vocabulaire terminal V_T qui commence toute chaîne qui se dérive de x (ε inclus), comme dans $x \Rightarrow * \ \mathbf{a}\beta$.

Enonçons deux propriétés qui vont servir en pratique.

Propriété n°1.2.1 :

si $\alpha \in V_T^*$, **First** $(\alpha x) = \{\alpha\}$

Propriété n°1.2.2 :

$\forall (x,y) \in (V_N \cup V_T)^{*2}$ **First**(xy) = **First**(x),
sauf dans le cas où $x \Rightarrow * \varepsilon$, on a alors :
First(xy) = **First**(x) $\cup$ **First**(y)

Définition des ensembles Initiaux :

$$\textbf{Init}\ (A) = \bigcup_{k=1}^{n} \textbf{First}\ (\alpha_k)$$

où : α_k est l'une des expansions de $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

convention : si $A \in V_T$, **Init** (A) = {A}

Propriété n°1.2.3 :

Dans le cas où il n'y a qu'une expansion pour A :
$A \rightarrow \alpha$
Nous avons : **Init** (A) = **First** (α)

Ces ensembles **Init** (A) permettent de rassembler les symboles terminaux qui se trouvent au début de tout mot dérivé par chacune des expansions de A ($A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$). Nous traitons un exemple complet plus loin.

1.3 Ensemble Follow

Seulement dans le cas où $A \in V_N$, on calcule l'ensemble suivant :

Follow(A) = { **a** $\in V_T$ / S $\Rightarrow^* \alpha Ax$ (où $\alpha \in V_T$, et **a** $\in$ **Init** (x)) }

Cet ensemble correspond aux symboles qui suivent les mots dérivés de A dans la grammaire. Ce sont les éléments terminaux **a** $\in V_T$ apparaissant immédiatement à droite de A dans toutes les chaînes contenant A, comme dans $S \Rightarrow \alpha A\mathbf{a}\beta$

Grâce aux ensembles **Init** et **Follow**, nous pouvons dire que nous disposons de deux familles de symboles qui encadrent les mots de la grammaire G. Cette remarque soulève le voile sur une stratégie pratique de reconnaissance des mots engendrés par une grammaire.

Enonçons une propriété calculatoire pratique, des **Follow** qui nous servira:

propriété n°1.3.1 :

si $A \rightarrow \alpha B$, $A \in V_N$, $B \in V_N$, $\alpha \in V_T^*$ **alors**
Follow (A) $\subset$ **Follow** (B)
fsi

1.4 Calcul des Follows à partir des First

Soit G une C-grammaire, G = (V_N,V_T,S,R).

Calcul des Follow(A) où $A \in V_N$:
Les règles contenant A en partie droite peuvent avoir d'une manière générale deux formes :
B $\rightarrow \alpha A\beta$ ou **B** $\rightarrow \alpha A$ ($B \in V_N$).

Pour toute règle de la forme **B** $\rightarrow \alpha A\beta$, (si $\varepsilon \notin$ **First** (β))

Follow (**A**) = **Follow** (**A**) $\cup$ **First** (β)

Pour toute règle de la forme **B** $\rightarrow \alpha A\beta$, (si $\varepsilon \in$ **First** (β))
et Pour toute règle de la forme **B** $\rightarrow \alpha A$

Follow (**A**) = **Follow** (**A**) $\cup$ **Follow** (**B**)

Exemple de calcul sur une C-grammaire G :

```
VT = {a, b}
VN = {A, S}
axiome: S
Règles :
```

```
1 : S → aAS
2 : S → b
3 : A → a
4 : A → bSA
```

On suppose en outre, que tous les mots de G se termineront par le symbole spécial '#' de fin de mot.

Calcul de Init(S)	
Il y a 2 expansions de S (Règle 1 et Règle 2) **Init**(S) = **First** (aAS) ∪ **First** (b) nous avons : **First** (b) = {b} nous avons : **First** (aAS) = {a}	Donc **Init**(S) = {a} ∪ {b}
Calcul de Init(A)	
Il y a 2 expansions de A (Règle 3 et Règle 4) **Init** (A) = **First** (a) ∪ **First** (bSA) nous avons : **First** (a) = {a} nous avons : **First** (bSA) = {b}	Donc : **Init** (A) = {a} ∪ {b}

De ces deux calculs nous concluons que : **Init (S) = Init (A) = {a,b}**

Passons au calcul des Follow à l'aide des **Init** que nous venons d'évaluer.

Calcul de Follow(S)	
S est l'axiome de la grammaire d'après la définition: **Follow** (S) = {#} Dans la règle 4 (**A** → bSA) d'après la définition: **Follow** (S) = **Follow** (S) ∪ **Init** (A)	D'où : **Follow (S) = { # , a , b }**
Calcul de Follow(A)	
Initialisation de **Follow** (A) = ∅ Dans la règle 4 (**S** → aAS) d'après la définition : **Follow** (A) = **Init** (S)	D'où : **Follow (A) = { a , b }**

Résultats obtenus	
VT = {a, b} VN = {A, S} axiome: S Règles : 1 : S → aAS 2 : S → b 3 : A → a 4 : A → bSA	**Init (S) = Init (A) = {a,b}** **Follow (S) = { # , a , b }** **Follow (A) = { a , b }**

En langage pratique nous pouvons dire que toute chaîne qui dérive de **A** comme de **S** commence soit par un symbole **a** ou par un symbole **b**, toute chaîne qui suit un mot dérivé de **A** commence par un symbole **a** ou par un symbole **b**, de même , enfin toute chaîne qui suit un mot dérivé de **S** commence par un symbole **a** ou par un symbole **b** ou un symbole **#.**

1.5 Calcul pour une grammaire arithmétique

Prenons un exemple plus pratique en utilisant une grammaire G_{exp} ambiguë des expressions arithmétiques, déjà proposée auparavant :

G_{exp} = (V_N,V_T,Axiome,Règles)
$\mathbf{V_T}$ ={ **0**, ..., **9**, **+**, **-**, **/**, *****, **)**, **(** }
$\mathbf{V_N}$ = { ⟨ Expr ⟩, ⟨ Nbr ⟩, ⟨ Cte ⟩, ⟨ Oper ⟩ }
Axiome : ⟨ Expr ⟩
Règles :
1 : ⟨ Expr ⟩ → ⟨ Nbr ⟩ | **(** ⟨ Expr ⟩ **)** | ⟨ Expr ⟩ ⟨ Oper ⟩ ⟨ Expr ⟩
2 : ⟨ Nbr ⟩ → ⟨ Cte ⟩ | ⟨ Cte ⟩ ⟨ Nbr ⟩
3 : ⟨ Cte ⟩ → **0** | **1** |...| **9**
4 : ⟨ Oper ⟩ → **+** | **-** | ***** | **/**

Calcul des ensembles Init

Provenant de la règle n°1
⟨Expr⟩ → ⟨Nbr⟩ | **(** ⟨Expr⟩ **)** | ⟨Expr⟩ ⟨Oper⟩ ⟨Expr⟩ D'après la définition des Init : **Init** (Expr) = **First** (Nbr) ∪ **First** ((Expr)) ∪ **First**(Expr Oper Expr) D'après la propriété 1.2.1 [*A = (Expr) est de la forme A = **aB** où **a** = (*] donc : **First** (**(** Expr **)**)={'**(**'} Comme ε ne dérive pas de Expr , nous avons : **First** (Expr Oper Expr) = **First** (Expr) Comme Nbr ∈ V_N possède deux expansions, son **First** n'est pas calculable immédiatement, il faut calculer son ensemble **Init** (Nbr).

Premier résultat obtenu
Init (Expr) = Init (Nbr) ∪{ **(** }

Provenant de la règle n°2
<Nbr> → <Cte> | <Cte> <Nbr> D'après la définition des Init : **Init** (Nbr) = **First** (Cte) ∪ **First**(Cte Nbr) D'après la propriété 1.2.2: **First**(Cte Nbr) = **First** (Cte) ∪ **First**(Nbr) Comme Cte ∈ V_N possède plusieurs expansions, son **First** n'est pas calculable immédiatement, il faut calculer son ensemble **Init** (Cte).

second résultat obtenu
Init (Nbr) = Init (Cte)

Nous terminons les calculs car il ne reste que des règles terminales ce qui donne des calculs de **First** immédiats.

Provenant de la règle terminale n°3
<Cte> → 0 | 1… | 9 D'après la définition des Init : **Init** (Cte) = **First** (0) ∪ ... ∪ **First**(9) = { 0,1, …, 9 }
troisième résultat obtenu
Init (Cte) = { 0,1, …, 9 }
Provenant de la règle terminale n°4
⟨Oper⟩ → + | - | * | / D'après la définition des Init : **Init** (Oper) = **First** (+) ∪ **First**(-) ∪ **First**(*) ∪ **First**(/) = { +, - , *, / }
quatrième résultat obtenu
Init (Oper) = { +, - , *, / }

En rassemblant les résultats calculés nous obtenons les **Init** de chaque élément de V_N de la grammaire G_{exp} des expressions arithmétiques

Init (Oper) = { +, - , * , / } **Init** (Cte) = { 0,1,...,9 } **Init** (Nbr) = { 0,1,...,9 } **Init** (Expr) = { 0,1,...,9,(}	Rappellons la signification pratique par exemple de l'ensemble **Init** (Expr) = { 0,1,...,9,(}: Toute expression dérivant de Expr commence par un chiffre de 0 à 9 ou bien commence par une parenthèse ouvrante '(' .

Calcul des ensembles Follow

Calcul de **Follow** (Expr)

Nous devons chercher dans toutes les règles de la grammaire celles qui contiennent **en partie droite le non terminal Expr**. Lorsque dans une règle de la forme " A → α<Expr>β " , Expr est en partie droite, en appliquant la définition, nous obtenons le fait que **Follow** (Expr) contient le First(β). Nous procédons donc ici à un calcul incrémental de l'ensemble **Follow** (Expr) par adjonctions successives des **First** qui le composent.

Provenant de la règle n°1
Les 2 premières expansions : ⟨Expr⟩ → ⟨Nbr⟩ ⟨Expr⟩ → **(** ⟨Expr⟩ **)** **Init** (‘**)**‘) = **First** (‘**)**’) ⊂ **Follow** (Expr) *d'après la définition* La troisième expansion : ⟨Expr⟩ → ⟨Expr⟩ ⟨Oper⟩ ⟨Expr⟩ **Init** (Oper) ⊂ **Follow** (Expr) *d'après la définition* Comme il n'y a pas dans G_{exp} d'autres règles contenant le symbole Expr en partie droite, donc le calcul du **Follow** (Expr) est terminé , **Follow** (Expr) ne contient que les ensembles **Init** (‘**)**‘) et **Init** (Oper) : **Follow** (Expr) = **Init** (‘**)**‘) ∪ **Init** (Oper)

Premier Follow obtenu
Follow (Expr) = { +, - , *, / ,) }

Le calcul est identique pour tous les autres follow

Calcul de **Follow** (Nbr) *(examen de toutes les parties droites contenant Nbr)*

Provenant de la règle n°1
⟨Expr⟩ → ⟨Nbr⟩ D'après la propriété 1.3.1 : **Follow** (Expr) ⊂ **Follow** (Nbr)
Provenant de la règle n°2
⟨ Nbr ⟩ →⟨Cte⟩ ⟨Nbr⟩ *rien de plus n'est ajouté.*

second Follow obtenu

Il n'y a pas d'autres règles contenant le symbole Nbr en partie droite, donc le calcul du **Follow** (Nbr) est terminé :

Follow (Nbr) = **Follow** (Expr) = {+, - , * , / ,) }

Calcul de **Follow** (Cte) *(examen de toutes les parties droites contenant Cte)*

Provenant de la règle n°2

⟨ Nbr ⟩ →⟨Cte⟩ ⟨Nbr⟩

D'après la propriété 1.3.1 :
Follow (Nbr) ⊂ **Follow** (Cte)

D'après la définition :
Init (Nbr) ⊂ **Follow** (Cte)

troisième Follow obtenu

Il n'y a pas d'autres règles contenant le symbole Cte en partie droite, donc le calcul du **Follow** (Cte) est terminé :

Follow (Cte) = **Follow** (Nbr) ∪ **Init** (Nbr) = {+, - , * , / ,) , 0 , 1 , … , 9 }

Calcul de **Follow** (Oper) *(examen de toutes les parties droites contenant Oper)*

Provenant de la règle n°1

⟨ Nbr ⟩ →⟨Cte⟩ ⟨Nbr⟩

D'après la définition :
Init (Expr) ⊂ **Follow** (Oper)

quatrième Follow obtenu

Il n'y a pas d'autres règles contenant le symbole Oper en partie droite, donc le calcul du **Follow** (Oper) est terminé : `Follow(Oper)= Init(Expr)= { 0,1,...,9,( }`

Follow (Oper) = **Init** (`Expr`) = { 0 , 1 , … , 9 , (}

Récapitulons les **Init et les Follow** de chaque élément de V_N de la grammaire G_{exp}	
Init (Expr) = { 0,1,...,9,(} **Init** (Nbr) = { 0,1,...,9 } **Init** (Cte) = { 0,1,...,9 } **Init** (Oper) = { +, - , * , / } **Follow** (Expr) = { +, - , *, / ,) } **Follow** (Nbr) = {+, - , * , / ,) } **Follow** (Cte) = {+, -, *, /,), 0, 1, … ,9 } **Follow** (Oper) = { 0 , 1 , … , 9 , (}	Rappellons la signification pratique par exemple des ensembles **Follow.** **Follow** (Expr): Tout mot suivant une expression dérivant de Expr commence par +, - , *, / ,). **Follow** (Cte): Tout mot suivant une constante dérivant de Cte commence par +, -, *, /,), 0, 1, … ,9 . **Follow** (Oper): Tout mot suivant un opérateur dérivant de Oper commence par 0 , 1 , … , 9 , (.

Nous pouvons savoir ainsi en consultant ces ensembles si les expressions suivantes sont correctes ou incorrectes dans G_{exp} :

> **12(5+3)** est incorrecte car la parenthèse ouvrante qui est le First de (5+3) n'est pas dans le Follow du nombre 12 (il n'y a pas de parenthèse ouvrante après un nombre).
>
> **12*)+2** est incorrecte car la parenthèse fermante qui est le First de)+2 n'est pas dans le Follow de l'opérateur * (il n'y a pas de parenthèse fermante après un opérateur).

1.6 Calcul pour une grammaire du mini-français

Soit G_{F1} une grammaire déjà étudiée au chapitre 6 d'un mini-français.

G = (V_N,V_T,S,R)
V_T ={**le, un, chat, chien, aime, poursuit, malicieusement, joyeusement, gentil, noir, blanc, beau, '.'**}
V_N = { ⟨*phrase*⟩, ⟨*GN*⟩, ⟨*GV*⟩, ⟨*Art*⟩, ⟨*Nom*⟩, ⟨*Adj*⟩, ⟨*Adv*⟩, ⟨*verbe*⟩ }
Axiome : ⟨ *phrase* ⟩
Règles :

```
1 : ⟨ phrase ⟩→⟨ GN ⟩⟨ GV ⟩⟨ GN ⟩ .
2 : ⟨ GN ⟩→⟨ Art ⟩⟨ Adj ⟩⟨ Nom ⟩
3 : ⟨ GN ⟩→⟨ Art ⟩⟨ Nom ⟩⟨ Adj ⟩
4 : ⟨ GV ⟩→⟨ verbe ⟩ | ⟨ verbe ⟩⟨ Adv ⟩
5 : ⟨ Art ⟩→ le | un
6 : ⟨ Nom ⟩→ chien | chat
7 : ⟨ verbe ⟩→ aime | poursuit
8 : ⟨ Adj ⟩→ blanc | noir | gentil | beau
9 : ⟨ Adv ⟩→ malicieusement | joyeusement
```

Nous livrons ci-après le calcul des ensembles **Init** et **Follow** sans le détailler car il s'agit de l'application à cet exemple de la même stratégie utilisée dans le cas de la grammaire précédente. Il est conseillé au lecteur de refaire le calcul lui-même à titre d'entraînement.

Indications sur le calcul des ensembles Init :

Init (Phrase) = **Init** (GN)
Init (GN) = **Init** (Art)
Init (GV) = **Init** (Verbe)
Init (Art) = {le,un}
Init (Nom) = {chat,chien}
Init (Verbe) = { aime, poursuit }
Init (Adj) = { gentil, noir, blanc, beau }
Init (Adv) = { malicieusement, joyeusement }

Récapitulatif :

Init (Phrase) = {le,un}
Init (GN) = {le,un}
Init (GV) = { aime, poursuit }
Init (Art) = {le,un}
Init (Nom) = {chat,chien}
Init (Verbe) = { aime, poursuit }
Init (Adj) = { gentil, noir, blanc, beau }
Init (Adv) = { malicieusement, joyeusement }

Indications sur le calcul des ensembles Follow :

Follow(Phrase) = {#} *fin de chaîne*
Follow(GN) = Init(GV) ∪ Init('.') *à partir de règle.1*
Follow(GV) = Init(GN) *à partir de règle.1*
Follow(Art) = Init(Adj) ∪ Init(Nom) *à partir de règles.2 et 3*
Follow(Nom) = Follow(GN) ∪ Init(Adj) *à partir de règles.2 et 3*
Follow(Verbe) = Follow(GV) ∪ Init(Adv) *à partir de règle 4*
Follow(Adj) = Follow(GN) ∪ Init(Nom) *à partir de règles.2 et 3*
Follow(Adv) = Follow(GV) ∪ Init(Adj) *à partir de règle 4*

Récapitulatif

Follow (Phrase) = {#}
Follow (GN) = { aime, poursuit, '.' }
Follow (GV) = {le,un}
Follow (Art) = { gentil, noir, blanc, beau, chat, chien }
Follow (Nom) = { gentil, noir, blanc, beau, aime, poursuit }
Follow (Verbe) = { le, un, malicieusement, joyeusement }
Follow (Adj) = { aime, poursuit, chat, chien, '.' }
Follow (Adv) = {le,un}

Voyons maintenant comment nous pouvons utiliser pratiquement ces ensembles Follow et Init (ou First) et dans quel cadre s'en servir. Le support de stratégie se dénomme l'analyse LL(1).

Nous décortiquons dans le paragraphe suivant un exemple pratique complet en soulignant les aspects méthodologiques généraux sous-jacents.

2. Grammaire LL(1) pour analyse déterministe

Les ensembles Init et Follow précédemment étudiés présentent un intérêt pratique dans l'analyse de mots d'une C-grammaire d'un " bon type ". C'est en vue d'une construction ultérieure systématique du filtrage du dialogue homme-machine que nous les avons présentés. En outre ils pourront aussi, comme nous le verrons lorsque nous aborderons ce thème, diriger notre programmation par plans d'action dans le guidage d'une saisie sur une interface. Dans le cas de dialogue avec l'utilisateur, c'est le concepteur du logiciel qui prévoit le " genre " de dialogue. Donc s'il dispose d'un outil rapide d'analyse du dialogue, il pourra dans la majorité des cas essayer d'élaborer une grammaire analysable rapidement sans alourdir sa tâche de programmation.

Nous avons choisi de présenter la technique la plus simple pour reconnaître les mots d'un langage dans le cas où la C-grammaire est de type LL(1).

La stratégie générale que nous adoptons est la suivante :

A chaque analyse du symbole en cours, il nous suffira de " regarder " le symbole suivant. Si la grammaire s'y prête (et nous allons donner les propriétés de telles grammaires), nous pourrons connaître à l'avance l'ensemble de tous les symboles (tous différents)pouvant se trouver après le symbole analysé. Chacun des symboles conduira à une branche d'analyse différente, le procédé est donc déterministe.

Une bonne C-grammaire (donc de type-2)qui se prête à ce genre d'analyse est dite analysable par la technique LL(1) ou plus brièvement appelée grammaire LL(1).

Nous possédons un mécanisme de construction d'analyseur de mots avec les automates d'états finis pour les grammaire de type-3. Le procédé LL(1) est un outil simple mais suffisamment intéressant pour fournir un cadre méthodique et introductif à d'autres développements. Avec cet outil de nombreux exemples efficaces et non triviaux peuvent être construits et programmés au niveau de l'initiation.

2.1 Une condition pratique d'analysabilité LL(1)

Nous donnons une condition nécessaire et suffisante posée comme définition pour qu'une C-grammaire soit LL(1). Cette CNS est un énoncé constructif qui va nous servir à vérifier rapidement si nous avons une grammaire LL(1) ou non.

CNS-LL(1):
$\forall A \in V_N$, $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots\ldots \mid \alpha_n$ $\forall (i,j) / i \neq j$: **Init**$(\alpha_i) \cap$ **Init**$(\alpha_j) = \varnothing$ **si** $\alpha_k \Rightarrow^* \varepsilon$ **alors** $\forall i / i \neq k$, **Init**$(\alpha_i) \cap$ **Follow**$(A) = \varnothing$ **fsi**

2.2 Méthode pratique de vérification

Afin de savoir si une C-grammaire donnée est LL(1) et pour appliquer la CNS précédente, nous construisons une table regroupant toutes les informations sur les Init et les Follow de chaque élément du vocabulaire auxiliaire de la grammaire.
Soient **A** $\rightarrow$ X1 |...| **X**n toutes les expansions de A, A $\in V_N$. Nous construisons le tableau suivant pour chaque élément A $\in V_N$:

Sym $\in V_N$	Init (Sym)				Follow(Sym)
	X_1	X_2	.....	X_n	
A	...	...		...	...

Où chaque colonne identifiée par $\mathbf{X}_k$ contient tous les symboles de l'ensemble **First**($\mathbf{X}_k$).

Puis, à l'aide de ce tableau, nous appliquons la CNS-LL1 :

- Pour un A, A $\in V_N$ fixé, les contenus de chacune des colonnes **X**k ne doivent pas avoir d'éléments communs.
- Si une expansion **X**k de A se dérive en ε ($\mathbf{X}_k \Rightarrow^* \varepsilon$), les contenus de toutes les autres colonnes **X**p (p ≠ k) ne doivent pas avoir d'éléments commun avec **Follow**(A).

Ci-après, les tables construites à partir des deux calculs précédents sur la grammaire des expressions arithmétiques et celle du mini-français :

	Init			Follow
Expr	**0,...,9**	**(**	**0,...,9**	**+ , * , -, /,)**
Nbr	**0,...,9**			**+ , * , -, /,)**
Cte	**0,...,9**			**+, *, -, /,), 0,...,9**
Oper	**+ , * , -, /**			**0,..., 9, (**

tableau pour la grammaire G_{exp}

Nous observons que G_{exp} n'est pas LL(1), car le premier **First** et le second **First** du symbole Expr ont une intersection non vide : **0,...,9.**

Remarque :

> Ceci est en particulier dû au fait que la récursivité gauche pose un problème pour ce genre d'analyse. Il y a deux **First** identiques.

Décrivons maintenant le tableau associé à la grammaire du mini-français notée plus haut G_{F1}:

	Init		Follow
Phrase	**le, un**		**#**
GN	**le, un**	**le, un**	**aime, poursuit, '.'**
GV	**aime, poursuit**	**aime, poursuit**	**le, un**
Art	**le, un**		**blanc, noir, gentil, beau, chat, chien**
Nom	**chien, chat**		**blanc, noir, gentil, beau, aime, poursuit**
Verbe	**aime, poursuit**		**le,un,malicieusement, joyeusement**
Adj	**blanc,noir,gentil,beau**		**aime, poursuit, chat, chien**
Adv	**malicieusement, joyeusement**		**le, un**

tableau pour la grammaire G_{F1}

Cette grammaire G_{F1} n'est pas LL(1) à cause de l'**Init** de GN ou de celui de GV, qui ont une intersection non vide de leurs **First**. Le problème est dû à la présence dans cette grammaire de règles non déterministes de la forme : $\mathbf{A} \rightarrow \alpha_1 B \mid \alpha_1 C$, (où $B \in V_N$ et $C \in V_N$).

De telles règles $\mathbf{A} \rightarrow \alpha_1 B \mid \alpha_1 C$ ne peuventt pas être analysées avec la stratégie d'observation du prochain symbole, puisque les deux expansions $\alpha_1 B$ et $\alpha_1 C$ débutent chacune par les mêmes symboles (ceux de **First** (α_1)).

Nous pouvons éliminer ce problème en construisant une autre grammaire en ajoutant un élément A' au vocabulaire auxiliaire de G_{F1} et en remplaçant les règles **A** $\rightarrow \alpha_1 B \mid \alpha_1 C$ par les deux règles suivantes (procédé classique en compilation appelé **factorisation**):

```
A  → α1A'
A' → B | C
```

(en espérant que **First** (A) et **First** (B) n'aient pas d'éléments communs)

Voyons enfin comment les outils que nous venons de considérer peuvent servir en programmation de l'analyse de phrases.

3.Analyser des phrases

3.1 Une grammaire LL(1) du mini-français

Voici G_{F2} une grammaire LL(1) obtenue à partir de G_{F1} par changement des règles selon l'idée de transformation précédente. Nous avons ajouté deux nouveaux symboles dans V_N: <LeNom> et <Suite>

$G_{F2} = (V_N, V_T, S, R)$
V_T ={le, un, chat, chien, aime, poursuit, malicieusement, joyeusement, gentil, noir, blanc, beau}
V_N = {⟨phrase⟩, ⟨GN⟩, ⟨GV⟩, ⟨Art⟩, ⟨Nom⟩, ⟨Adj⟩, ⟨Adv⟩, ⟨verbe⟩,⟨LeNom⟩, ⟨Suite⟩}
Axiome : ⟨phrase⟩
Règles :

```
1 :⟨ phrase ⟩→⟨ GN ⟩⟨ GV ⟩⟨ GN ⟩.
2 :⟨ GN ⟩→⟨ Art ⟩⟨ LeNom ⟩
3 :⟨ LeNom ⟩ →⟨ Adj ⟩⟨ Nom ⟩ | ⟨ Nom ⟩⟨ Adj ⟩
4 :⟨ GV ⟩→⟨ verbe ⟩⟨ Suite ⟩
5 :⟨ Suite ⟩ →⟨ GN ⟩ | ⟨ Adv ⟩⟨ GN ⟩
6 :⟨ Art ⟩→ le | un
7 :⟨ Nom ⟩→ chien | chat
8 :⟨ verbe ⟩→ aime | poursuit
9 :⟨ Adj ⟩→ blanc | noir | gentil | beau
10 :⟨ Adv ⟩→ malicieusement | joyeusement
```

En calculant comme précédemment les Init et les Follow, nous obtenons le tableau récapitulatif suivant:

	Init		Follow
Phrase	**le, un**		**#**
GN	**le, un**		**aime, poursuit, '.'**
LeNom	**blanc, noir, gentil, beau**	**chien, chat**	**aime, poursuit, '.'**
GV	**aime, poursuit**		**'.'**
Suite	**le, un**	**malicieusement, joyeusement**	**'.'**
Art	**le, un**		**blanc, noir, gentil, beau, chat, chien**
Nom	**chien, chat**		**blanc, noir, gentil, beau, aime, poursuit, '.'**
Verbe	**aime, poursuit**		**le,un,malicieusement, joyeusement**
Adj	**blanc, noir, gentil, beau**		**aime, poursuit, chat, chien, '.'**
Adv	**malicieusement, joyeusement**		**le, un**

ci-dessus un tableau pour la grammaire G_{F2}

Elle vérifie la CNS-LL1, cette grammaire G_{F2} est LL(1).
Nous indiquons ensuite un moyen destiné à effectuer la reconnaissance manuelle tout d'abord, puis par programme en Delphi par exemple, uniquement dans le cas de la grammaire LL(1) G_{F2} du mini-français. Cela pourra inciter l'étudiant à aller chercher dans les ouvrages spécialisés les méthodes générales mettant en œuvre ces techniques de reconnaissance.

3.2 Schémas d'algorithmes associés à G_{F2}

Nous supposons disposer d'un moyen d'extraire dans la phrase le symbole à analyser. Il sera mis dans la variable notée " **SymLu** ". Nous découpons notre travail d'analyse en blocs comme nous l'avions découpé lors de l'utilisation d'une grammaire en mode génération de phrases. Chaque bloc syntaxique est associé à un élément du vocabulaire auxiliaire V_N dans G_{F2}.
La démarche descendante reste semblable à la " programmation par la syntaxe" déjà vue et assure une cohérence pédagogique sur la méthode de travail adoptée.

Nous supposons disposer de deux outils supplémentaires pour effectuer notre analyse :

- un outil nommé " **Symsuivant** " qui met le prochain symbole de la phrase dans " **SymLu** ",

- un outil " **Erreur** " de signalement d'erreur dès qu'une erreur d'analyse est détectée. L'outil " Erreur " arrête en même temps tout le processus de reconnaissance.

Voici une écriture algorithmique des différents blocs d'analyse associés aux éléments de V_N dans G_{F2}. Cette écriture permet en l'appliquant à une phrase de G_{F2}, de valider ou non son analyse (de la reconnaître). Notre stratégie est basée sur les **Init** de chaque symbole de V_N utilisés comme ensembles de prédiction sur l'unique " bon chemin " à prendre dans la suite des règles.

Bloc Analyser Phrase :

```
si SymLu ∈ Init(GN) alors
 Analyser GN ;
 si SymLu ∈ Init(GV) alors
  Analyser GV ;
  si SymLu ≠ '.' Alors Erreur fsi
 sinon Erreur
 fsi
sinon Erreur
fsi
```

Bloc Analyser GN :

```
si SymLu ∈ Init(Art) alors
 Analyser Art;
 si SymLu ∈ Init(LeNom) alors
  Analyser LeNom;
 sinon Erreur
 fsi
sinon Erreur
fsi
```

Bloc Analyser GV :

```
si SymLu ∈ Init(Verbe) alors
 Analyser Verbe;
 si SymLu ∈ Init(Suite) alors
  Analyser Suite;
 sinon Erreur
 fsi
sinon Erreur
fsi
```

Bloc Analyser Suite :

```
si SymLu ∈ Init(GN) alors
 Analyser GN;
sinon
 si SymLu ∈ Init(Adv) alors
  Analyser Adv;
  si SymLu ∈ Init(GN) alors
   Analyser GN;
  sinon Erreur
  fsi
 sinon Erreur
 fsi
fsi
```

Bloc Analyser LeNom :

```
si SymLu ∈ Init(Adj) alors
 Analyser Adj;
 si SymLu ∈ Init(Nom) alors
  Analyser Nom
 sinon Erreur
 fsi
sinon
 si SymLu ∈ Init(Nom) alors
  Analyser Nom;
  si SymLu ∈ Init(Adj) alors
   Analyser Adj;
  sinon Erreur
  fsi
 sinon Erreur
 fsi
fsi
```

Bloc Analyser Art :

```
si SymLu ∈ {le,un } alors
  Symsuivant
sinon Erreur
fsi
```

Bloc Analyser Nom :

```
si SymLu ∈ {chat,chien } alors
 Symsuivant
sinon Erreur
fsi
```

Bloc Analyser Verbe :

```
si SymLu ∈ {aime, poursuit }
  alors   Symsuivant
sinon Erreur
fsi
```

Bloc Analyser Adj :	Bloc Analyser Adv :
si SymLu ∈ {beau, blanc, noir, gentil } **alors** Symsuivant **sinon** Erreur **fsi**	**si** SymLu ∈ {malicieusement, joyeusement } **alors** Symsuivant **sinon** Erreur **fsi**

3.3 Procédures Pascal-Delphi associées aux blocs dans G_{F2}

Afin de clore cette ouverture sur la reconnaissance, nous traduisons en pascal les spécifications précédentes sur les blocs d'analyse. A chaque bloc d'analyse nous faisons correspondre une procédure pascal-Delphi en nous inspirant des choix effectués lors de l'écriture d'un générateur de phrase du mini-français.

Les structures de données :
Nous disposons d'un type **liste** obtenu à partir d'une unité de liste linéaire de chaînes.
Le type **liste** sert à stocker des éléments de V_T qui sont tous des **string**.

```
var
tnom,tadjectif,tarticle,tverbe,tadverbe:liste;
{toutes ces listes contiennent les symboles de V_T }

Init_Grp_Nom, Init_LeNom, Init_Grp_Verbal,Init_Suite:liste;
{toutes ces listes contiennent les symboles des Init}
Symlu:string; {le symbole lu}
PhraseLue,Copiephrase:string; {la phrase à analyser et sa copie}

Les outils de base :
procedure Symsuiv(var Sym:string);
{l'outil Symsuivant, le paramètre Sym
contient le symbole extrait}
begin
 if Copiephrase<>'' then
  if Copiephrase='.' then Sym:='.'
  else
  begin
   Sym:=copy(Copiephrase,1,pos(' ',Copiephrase)-1);
   delete(Copiephrase,pos(Sym,Copiephrase),length(sym)+1)
  end;
end;

procedure Erreur; {outil Erreur}
begin
 writeln('Erreur');
 readln ;
 halt
end;

function AppartientA(Sym:string;Ensemble:liste):boolean;
{teste l'appartenance à un Init donné du symbole Sym }
```

```
begin
 AppartientA:=False;
 if Test (Ensemble,Sym) then AppartientA:=True
end;
```

Traduction de chacun des blocs d'analyse

```
procedure Nom; {Bloc Analyser Nom}
begin
 if AppartientA(Symlu,tnom) then
  symsuiv(Symlu);
end;
```

```
procedure Adjectif; {Bloc Analyser Adj}
begin
 if AppartientA(Symlu,tadjectif) then
  symsuiv(Symlu);
end;
```

```
procedure Adverbe; {Bloc Analyser Adv}
begin
 if AppartientA(Symlu,tadverbe) then
  symsuiv(Symlu);
end;
```

```
procedure Verbe; {Bloc Analyser Verbe}
begin
 if AppartientA(Symlu,tverbe) then
  symsuiv(Symlu);
end;
```

```
procedure LeNom; {Bloc Analyser LeNom }
begin
 if AppartientA(Symlu,tadjectif) then
 begin
  Adjectif;
  if AppartientA(Symlu,tnom) then
   Nom
  else erreur
 end
 else
 if AppartientA(Symlu,tnom) then
 begin
  Nom;
  if AppartientA(Symlu,tadjectif) then
   Adjectif
  else erreur
 end
 else erreur
end;
```

```
procedure Grp_Nom; {Bloc Analyser GN}
begin
 if AppartientA(Symlu,tarticle) then
 begin
  Article;
  if AppartientA(Symlu, Init_LeNom) then
```

```
   LeNom
  else erreur
 end
 else erreur;
end;
```

```
procedure Suite; {Bloc Analyser Suite}
begin
 if AppartientA(Symlu,Init_Grp_Nom) then
  Grp_Nom
 else
 if AppartientA(Symlu,tadverbe) then
 begin
  Adverbe;
  if AppartientA(Symlu,Init_Grp_Nom) then
   Grp_Nom
  else erreur
 end
 else erreur
end;
```

```
procedure Grp_Verbal; { Bloc Analyser GV }
begin
 if AppartientA(Symlu,tverbe) then
 begin
  Verbe;
  if AppartientA(Symlu,Init_Suite) then
   Suite
  else erreur
 end
 else erreur
end;
```

```
procedure phrase; {Bloc Analyser Phrase }
begin
 if AppartientA(Symlu,Init_Grp_Nom) then
 begin
  Grp_Nom;
  if AppartientA(Symlu,Init_Grp_Verbal) then
  begin
   Grp_Verbal;
   if Symlu <>'.' then erreur
   else writeln('Phrase reconnue !')
  end
  else erreur
 end
 else erreur;
end;
```

Chapitre 7.3 interaction et pilotage

interface en mini français

Plan du chapitre:

1. Un peu plus loin avec l'interaction et le pilotage

1.1 Reconnaissance syntaxique du dialogue
1.2 Saisie dirigée par la syntaxe : principe
1.3 Utilisation du TAD grammaire graphique pour la saisie

2. Une méthode de construction

2.1 Tableau de traduction de Vt en diagrammes événementiels
2.2 Tableau de traduction de Vn en schémas LDFA
2.3 Interface de saisie du mini-français

1. Un peu plus loin avec l'interaction et le pilotage

Nous allons nous servir dans ce paragraphe, de ce que nous connaissons sur la programmation par les grammaires pour piloter les actions de dialogue avec l'utilisateur.

1.1 Reconnaissance syntaxique du dialogue

Nous supposons que le dialogue peut être spécifié par une grammaire (ceci est d'autant plus vrai que c'est le programmeur qui décide du genre de dialogue à instaurer dans son interface, il lui est donc loisible de définir une " bonne " grammaire pour le dialogue entre son logiciel et l'utilisateur).

Nous recensons deux catégories de dialogue :

> Soit l'on guide pas à pas l'utilisateur dans la saisie et il ne peut entrer que de l'information syntaxiquement correcte. Nous dénommerons cette catégorie sous le vocable " ***saisie dirigée par la syntaxe*** ".
>
> Soit l'utilisateur est libre de saisir de l'information et l'on lance une vérification de la syntaxe dès la fin de la saisie (ce qui se passe lorsque vous utilisez un compilateur qui vérifie votre programme source après que vous l'avez tapé). Cette deuxième façon de faire nécessite la construction d'un ***analyseur syntaxique*** (plus ou moins complexe selon que la grammaire est de type 2 ou 3 et selon sa classe d'analyse).

Le deuxième mode de guidage est classique et ressort de ce qui a été vu dans les chapitres précédents. Nous nous intéressons plutôt au premier mode de saisie qui est en fait plus aisé à programmer.

1.2 Saisie dirigée par la syntaxe : principe

Ce genre de saisie est le plus simple à mettre en œuvre en initiation, et donne des résultats immédiats ; en particulier, il se prête bien au prototypage avec un RAD visuel. Le programmeur peut élaborer un prototype (squelette d'IHM) de son interface, le faire fonctionner et le tester d'une manière autonome sans avoir besoin d'écrire le code interne complet. La partie visuelle et événementielle du RAD permet de construire, de tester et de modifier rapidement l'interface.

Le principe général de conception est le suivant :

> L'interface présente à l'utilisateur et pour chaque nouveau plan d'action, tous les choix admissibles pour le passage au plan d'action suivant. Ceci se traduira pour la saisie d'un texte par la présentation à l'utilisateur des symboles terminaux actuellement possibles pour construire son dialogue et au masquage de tous les autres. Nous introduisons ici la notion de **plan d'action** matérialisé par un ensemble d'états des objets de l'interface combiné avec des actions sur ces objets.

Nous nous servons d'une grammaire décrite par ses diagrammes syntaxiques pour spécifier les étapes de passage d'un plan d'action à l'autre (la grammaire peut être de type 3 ou de type 2 mais LL(1)).

1.3 Utilisation du TAD grammaire graphique pour la saisie

Nous rappelons le **TAD** générique ***Diag*** de grammaire graphique déjà défini pour spécifier de façon abstraite et graphique les règles d'une C-grammaire.

TAD : Diag
utilise : **VT**, **VN** *// les vocabulaires de la grammaire*
opérations :
t_1 : → Diag
t_2 : VT ∪ VN → Diag
t_3 : VN → Diag
t_4 : VT ∪ VN → Diag
t_5 : (VT ∪ VN)**n** → Diag
t_6 : (VT ∪ VN)**2** → Diag
t_7 : (VT ∪ VN)**2** → Diag
• : Diag x Diag → Diag

Axiomes :
la loi • est associative *(concaténation de diagrammes)*
$\forall t_i \in$ Diag $(i \geq 1)$ / $t_1 \bullet t_i = t_i \bullet t_1$ (t_1 *élément neutre)*
$t_i(A) \bullet t_k(B) = t_i(A)t_k(B)$ *(méthode de construction des diagrammes)*

Nous proposons au lecteur une spécification opérationnelle (plus concrète) de chaque diagramme de règle du **TAD** ***Diag*** à l'aide de diagrammes événementiels (les conventions utilisées sont celles qui ont été indiquées lors de la définition du graphe événementiel). Cette spécification a pour but de fournir un outil méthodique de construction d'une série de plans d'action (activation-désactivation) afin d'implanter une saisie dirigée par la syntaxe.

Les opérateurs t_k représentent pour notre interface des **plans d'action élémentaires**. Un plan d'action général est constitué d'une **combinaison de plans d'action élémentaires**.

Comme notre souci est de rester pratique, chaque diagramme événementiel associé à un opérateur $\mathbf{t_i}$, sera traduit par un exemple en Delphi.

> Nous avons choisi d'implanter les objets événementiels par des boutons de la classe des TButton.
>
> L'action d'activation ou de désactivation est implantée par la variation de la propriété booléenne de masquage " enabled " de l'objet TButton.

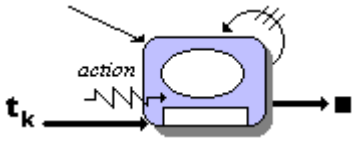
Remarque :

Nous aurions pu aussi utiliser une autre propriété booléenne de masquage des TButton, la propriété

" visible ".

Notes d'implantation en Delphi

A partir d'un diagramme événementiel général comme ci-dessous :



où :

t_k ⟶ = entrée dans le plan d'action t_k

⟶■ = passer au plan d'action suivant

Nous implanterons en Delphi:

action "↯" = " événement clic du bouton"
activation "⟶" = " NomduBouton.enabled := true "
désactivation "⟶" = " NomduBouton.enabled :=false "
gestionnaire d'action = gestionnaire d'événement Clic du bouton.

Avec comme apparence visuelle pour l'activation/désactivation :

Propriété enabled :=true (activable)

Propriété enabled :=false(inactivable)

L'exécution de plans d'action de saisie correspond dans ce cas essentiellement à masquer / démasquer des boutons utilisables.

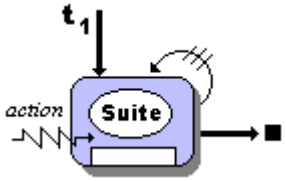
2. Une méthode pratique de construction

Nous proposons une méthode de construction de plans d'action en suivant **la syntaxe d'une grammaire LL(1)**. Nous associons des diagrammes événementiels et des schémas de plan d'action algorithmique aux diagrammes syntaxiques de la grammaire. L'implantation des plans d'action sera exécutée en Delphi.

2.1 Tableau de traduction de Vt en diagrammes événementiels

Cette traduction est valide pour des éléments A $\in V_T$et B $\in V_T$. L'élément " Suite " (associé à la règle vide) représente un événement permettant de passer d'un plan d'action au suivant sans avoir de choix terminal à proposer à l'utilisateur.

Opérateur t_1 :

Diagramme de règle	*Diagramme événementiel associé*
t_1 = A: →	

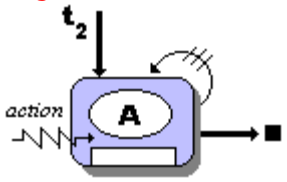
Implantation en Delphi du diagramme t_1 :

Au moment où le plan d'action t_1 est exécuté, le bouton Suite est activé.
Une action clic appelle le gestionnaire de l'événement clic du bouton suite.

Le code du gestionnaire du bouton **suite** est :

Suite.enabled :=false ; {*le bouton se désactive*} Suite
AfficherPlanSuivant ; {*exécution du plan suivant*}

Opérateur t_2 :

Diagramme de règle	*Diagramme événementiel associé*
$t_2(A)$ = B: —A→	

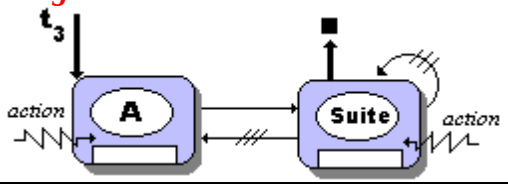
Implantation en Delphi du diagramme t_2 :

Au moment où le plan d'action t_2 est exécuté, le bouton A est activé.
Une action clic appelle le gestionnaire de l'événement clic du bouton A.

Le code du gestionnaire du bouton A est :

A.enabled :=false ; {*le bouton se désactive*} A
Saisie(A) ; {*saisie de l'information*}
AfficherPlanSuivant ; {*exécution du plan suivant*}

Opérateur t_3 :

Diagramme de règle	*Diagramme événementiel associé*
$t_3(A)$ = B: —A→	

Implantation en Delphi du diagramme t_3 :

Au moment où le plan d'action t_3 est exécuté, le bouton A est activé, le bouton Suite n'est pas activé.
Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A**.

Le code du gestionnaire du bouton **suite** est :

Suite.enabled :=false ; {*le bouton se désactive*} Suite
A.enabled :=false ; {*bouton A désactivé*} A
AfficherPlanSuivant ; {*exécution du plan suivant*}

Le code du gestionnaire du bouton **A** est :

Suite.enabled :=true ; {*bouton suite activé*} Suite
Saisie(A) ; {*saisie de l'information*}

Opérateur t_4 :

Diagramme de règle

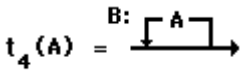
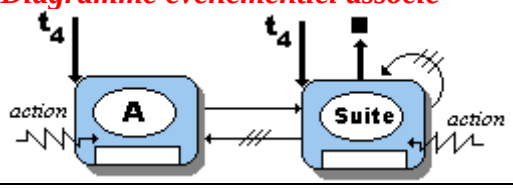


Diagramme événementiel associé



Implantation en Delphi du diagramme t_4 :

Au moment où le plan d'action t_4 est exécuté, le bouton A est activé, le bouton Suite est activé aussi.

Une action clic sur le bouton **suite** appelle le gestionnaire de l'événement clic du bouton suite. Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A**.

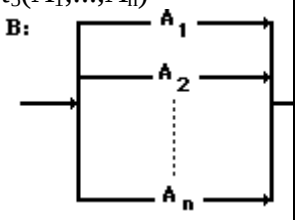
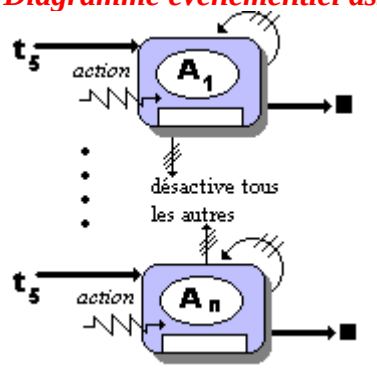
Le code du gestionnaire du bouton **suite** est :

Suite.enabled :=false ; {*le bouton se désactive*} Suite
A.enabled :=false ; {*bouton A désactivé*} A
AfficherPlanSuivant ; {*exécution du plan suivant*}

Le code du gestionnaire du bouton **A** est :

Suite.enabled :=true ; {*bouton suite activé*} Suite
Saisie(A) ; {*saisie de l'information*}

Opérateur t_5 :

Diagramme de règle	*Diagramme événementiel associé*
$t_5(A_1,...,A_n)$ = 	

Implantation en Delphi du diagramme t_5 :

Au moment où le plan d'action t_5 est exécuté, les boutons A1,...,An sont activés.

Une action clic sur un des boutons $\mathbf{A_k}$ appelle le gestionnaire de l'événement clic du bouton $\mathbf{A_k}$.

Le code du gestionnaire de chaque bouton $\mathbf{A_k}$ est :

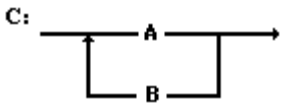
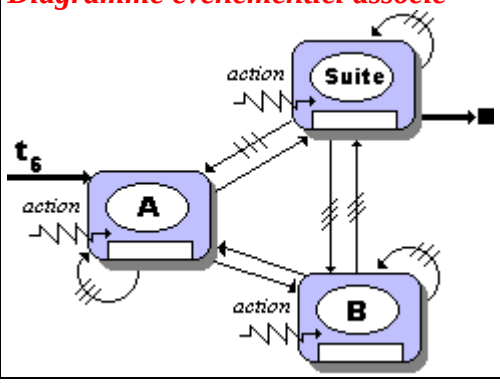
DésactiverTous; {*tous les boutons sont désactivés*} A1

...etc... An

Saisie(Ak) ; {*saisie de l'information de A_k*}

AfficherPlanSuivant ; {*exécution du plan suivant*}

Opérateur t_6 :

Diagramme de règle	*Diagramme événementiel associé*
$t_6(A,B)$= 	

Implantation en Delphi du diagramme t_6 :

Au moment où le plan d'action t_6 est exécuté, le bouton A est activé, les boutons Suite et B ne sont pas activés.
Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A** qui agit sur **suite** et **B**. Une action clic sur le bouton **B** appelle le gestionnaire de l'événement clic du bouton **B** qui agit sur **A** et **suite**. Une action clic sur le bouton **suite** appelle le gestionnaire de l'événement clic du bouton **suite** qui agit sur **A** et **B**.

Le code du gestionnaire du bouton **suite** est :

Suite.enabled :=false ; {*le bouton se désactive*} Suite
A.enabled :=false ; {*bouton A désactivé*} A
B.enabled :=false ; {*bouton B désactivé*} B
AfficherPlanSuivant ; {*exécution du plan suivant*}

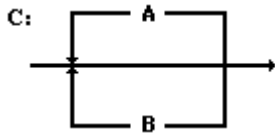
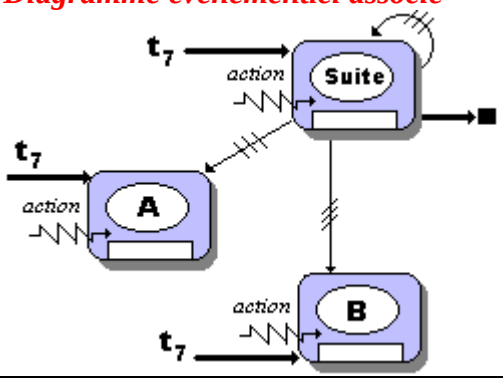
Le code du gestionnaire du bouton **A** est :

Suite.enabled :=true ; {*bouton suite activé*} Suite
B.enabled :=true ; {*bouton B activé*} B
A.enabled :=false ; {*bouton A désactivé*} A
Saisie(A) ; {*saisie de l'information de A*}

Le code du gestionnaire du bouton **B** est :

Suite.enabled :=false ; {*bouton suite activé*} Suite
B.enabled :=false ; {*bouton B activé*} B
A.enabled :=true ; {*bouton A désactivé*} A
Saisie(B) ; {*saisie de l'information de B*}

Opérateur t_7 :

Diagramme de règle	*Diagramme événementiel associé*
t_7(A,B)= 	

Implantation en Delphi du diagramme t_7 :

Au moment où le plan d'action t_7 est exécuté, les boutons A, Suite et B sont activés.
Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A**. Une action clic sur le bouton **B** appelle le gestionnaire de l'événement clic du bouton **B**. Une action clic sur le bouton **suite** appelle le gestionnaire de l'événement clic du bouton **suite** qui agit sur **A** et **B**.
Le code du gestionnaire du bouton **suite** est :

```
Suite.enabled :=false ; {le bouton se désactive}
A.enabled :=false ; {bouton A désactivé}
B.enabled :=false ; {bouton B désactivé}
AfficherPlanSuivant ; {exécution du plan suivant}
```

Le code du gestionnaire du bouton **A** est :

```
Saisie(A) ; {saisie de l'information de A}
```

Le code du gestionnaire du bouton **B** est :

```
Saisie(B) ; {saisie de l'information de B}
```

2.2 Tableau de traduction de V_N en schémas LDFA

Nous nous préoccupons maintenant des plans associés à des éléments du vocabulaire non terminal V_N de notre grammaire supposée être LL(1).

Le passage d'un plan d'action à un autre est conditionné par les symboles de V_t qui doivent être présentés à l'utilisateur dans le nouveau plan d'action. Or nous savons que l'ensemble Init(A) de l'élément A de V_N d'une grammaire nous fournit tous les symboles possibles d'un plan d'action associé à l'élément A. Puis nous ferons fonctionner notre grammaire en mode générateur. Au lieu d'engendrer aléatoirement des phrases du langage nous engendrons des phrases correctes où les choix des éléments terminaux ont été effectués par l'utilisateur. En partant de cette remarque, nous pouvons construire des schémas généraux écrits en langage d'algorithme (LDFA) décrivant le fonctionnement d'un plan d'action associé à un élément A, $A \in V_N$.

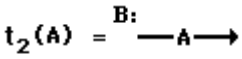
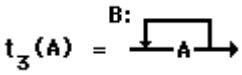
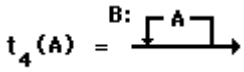
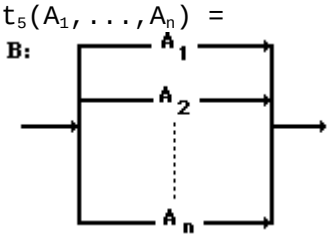
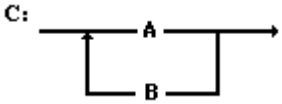
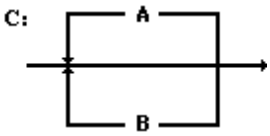
Le tableau avec ses objets et outils de base :
Nous supposons disposer d'un moyen d'activer (présenter à l'utilisateur) tous les symboles de Init(A) afin de ne lui laisser que ces choix possibles : **Activer(Init(A)).**

Nous supposons disposer d'un moyen de désactiver tous les symboles de Init(A) dès que l'utilisateur a fait son choix : **Désactiver(Init(A)).**

Le symbole **suite** a la même signification qu'au paragraphe précédent, il sert à passer au plan d'action suivant.

Le symbole **action** indique quelle est l'action que vient d'effectuer l'utilisateur sur l'interface.

Nous en déduisons le tableau de traduction de V_N en schéma Algorithmique LDFA :

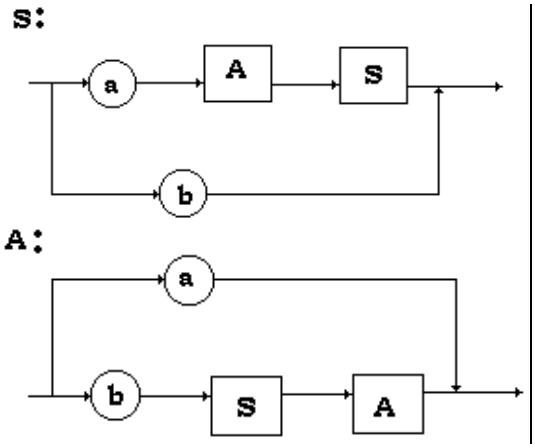
Diagramme de règle	Schéma LDFA associé
Opérateur t_2 $t_2(A)$ = 	Activer (Init(A)) Plan A ; Désactiver (Init(A))
Opérateur t_3 $t_3(A)$ = 	Activer (Init(A)) **Répéter** Plan A ; Activer (Init(Suite)) **jusquà** action = Suite ; Désactiver (Init(A))
Opérateur t_4 $t_4(A)$ = 	Activer (Init(A)) ; Activer (Suite) ; **Tantque** action ≠ Suite **faire** Plan A ; Activer (Suite) **Ftant** ; Désactiver (Init(A))
Opérateur t_5 $t_5(A_1,...,A_n)$ = 	Activer (Init(A_1)); Activer (Init(A_2)) ; .... **si** action ∈ Init(A_1) **alors** Plan A_1 **sinon si** action ∈ Init(A_2) **alors** Plan A_2 **sinon si** .... **fsi** ; Désactiver (Init(A_1)) ; ....
Opérateur t_6 $t_6(A,B)$= 	**répéter** Activer (Init(A)) ; Plan A ; Activer (Init(B)) ; Activer (Init(Suite)) ; **si** action ∈ Init(B) **alors** Plan B **fsi** **jusquà** action = Suite
Opérateur t_7 $t_7(A,B)$= 	Activer (Init(A)) ; Activer (Init(B)) ; Activer (Suite) ; **Tantque** action ∈ Init(B) ∪ Init(A) **faire** **si** action ∈ Init(B) **alors** Plan B **sinon** Plan A **fsi** ; Activer (Init(A)) ; Activer (Init(B)) ; Activer (Suite) ; **Ftant** ;

Application à deux exemples

Exemple-1

Soit la grammaire G déjà vue lors de l'étude des First et des Follow :

G:
VT = {a, b}
VN = {A, S}
<u>axiome</u>: S



<u>Règles</u> :

1 : S → aAS

2 : S → b

3 : A → a

4 : A → bSA

Cette grammaire est LL(1) car nous avons déjà calculé les Init de A et de S :

InitA) = Init(S) = {a,b}

En nous servant du tableau de traduction précédent écrivons les plans d'action associés aux deux éléments de V_N soient A et S.

<u>Plan A :</u> **Activer(a) ;** **Activer(b) ;** **Saisie(SymLu) ;** **si action = a alors** **désactiver(a)** **sinon** **désactiver(b);** **Plan S;** **Plan A** **fsi** **désactiver(a) ;** **désactiver(b) ;**	**<u>Plan S :</u>** **Activer(a) ;** **Activer(b) ;** **Saisie(SymLu) ;** **si action = a alors** **désactiver(a) ;** **Plan A ;** **Plan S** **sinon** **désactiver(b);** **fsi ;** **désactiver(a) ;** **désactiver(b) ;**

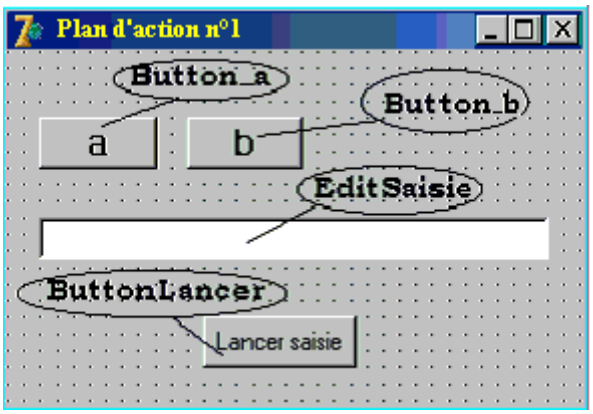
Implantation avec Delphi

- `Nous choisissons d'utiliser deux TButton pour les éléments terminaux " a "(Button_a) et " b "(Button_b), la saisie a lieu dans un TEdit (EditSaisie).`

- Les actions activer-désactiver sont implantées par la propriété enabled des Tbutton :

procedure Activer(Elt:TButton); **begin** Elt.enabled:=true; **end**;	**procedure** Desactiver(Elt:TButton); **begin** Elt.enabled:=false; **end**;

L'interface retenue est la suivante :



- Nous n'utilisons pas l'élément **suite**.
- Le symbole **action** est implanté par un drapeau booléen " ActionFaite " indiquant si l'utilisateur a effectué une action oui ou non.
- Nous avons conservé le mode séquentiel avec interruption afin de pouvoir bénéficier de la méthode de programmation par syntaxe en mode génération. La saisie du caractère dans **SymLu** est donc programmée selon une boucle d'attente infinie qui ne se libère que lorsque l'utilisateur a clické sur l'un des choix possibles.

```
procedure Attendre;
begin
  if not ActionFaite then
  begin
   repeat
      Application.ProcessMessages
   until ActionFaite ;
   ActionFaite:=false
  end
end{Attendre};

procedure saisie(ch:string);
begin
 if ActionFaite=false then
  Form1.Edit_saisie.text:= Form1.Edit_saisie.text+' '+ch
 else ActionFaite:=false
end{saisie};

procedure AttenteSaisie;
begin
 Attendre;
 saisie(SymLu);
end{AttenteSaisie};
```

Chacun des plans est implanté selon une procédure:

```
procedure Plan_A;
begin
 with Form1 do
 begin
  Activer(Button_a);
  Activer(Button_b);
  AttenteSaisie;
  if  SymLu='a' then
   Desactiver(Button_a)
  else
  begin
    Desactiver(Button_b);
    Plan_S;
    Plan_A
  end;
  Desactiver(Button_a);
  Desactiver(Button_b)
 end
end;
```

```
procedure Plan_S;
begin
 with Form1 do
 begin
  Activer(Button_a);
  Activer(Button_b);
  AttenteSaisie;
  if  SymLu='a' then
  begin
   Desactiver(Button_a);
   Plan_A;
   Plan_S
  end
  else
    Desactiver(Button_a);
  Desactiver(Button_b)
end
end;
```

Lorsque l'utilisateur sélectionne un bouton par un clic, le symbole SymLu contient la valeur du choix effectué.

```
procedure TForm1.Button_aetbClic(Sender: TObject);
begin
 ActionFaite:=true;
 Symlu:=TButton(Sender).caption
end;
```

Le bouton lancer la saisie appelle le plan associé à l'axiome S.

```
procedure TForm1.ButtonLancerClic(Sender: TObject);
begin
 RAZTout;
 ButtonLancer.enabled:=false;
 Plan_S;
 PlanSuivant;
end;
```

Il appelle aussi à la fin la procédure PlanSuivant qui sert à passer au plan d'action suivant (dans l'exemple le plan suivant est l'état initial, il suffit de réactiver le bouton lancer).

```
procedure PlanSuivant;
begin
 Form1.ButtonLancer.enabled:=true;
end;
```

Exemple-2 *Interface de saisie du mini-français*

La grammaire choisie est la grammaire LL(1) GF2 du mini-français déjà étudiée.

$G_{F2} = (V_N,V_T,S,R)$
V_T ={le, un, chat, chien, aime, poursuit, malicieusement, joyeusement, gentil, noir, blanc, beau, '.'}
V_N = { ⟨phrase⟩, ⟨GN⟩, ⟨GV⟩, ⟨Art⟩, ⟨Nom⟩, ⟨Adj⟩, ⟨Adv⟩, ⟨verbe⟩, ⟨LeNom⟩, ⟨Suite⟩ }

Axiome : <phrase>
Règles :
1 : ⟨phrase⟩→⟨GN⟩⟨GV⟩.
2 : ⟨GN⟩→ ⟨Art⟩⟨LeNom⟩
3 : ⟨LeNom⟩ →⟨Adj⟩⟨Nom⟩ | ⟨Nom⟩⟨Adj⟩
4 : ⟨GV⟩→ ⟨verbe⟩⟨Suite⟩
5 : ⟨Suite⟩→ ⟨GN⟩ | ⟨Adv⟩⟨GN⟩
6 : ⟨Art⟩→ le | un
7 : ⟨Nom⟩→ chien | chat
8 : ⟨verbe⟩→ aime | poursuit
9 : ⟨Adj⟩→ blanc | noir | gentil | beau
10: ⟨Adv⟩→ malicieusement | joyeusement

Nous laissons le lecteur écrire les diagrammes syntaxiques obtenus à partir des règles précédentes.

Afin que le lecteur puisse bien se pénétrer de la similitude des démarches entre les blocs d'analyse avec la saisie par plan d'action, nous lui livrons ci-dessous deux plans et les blocs correspondants, il continuera à écrire de la même façon ceux des autres éléments de V_N.

Premier plan d'action celui de l'axiome < phrase>

Bloc Analyser Phrase : **si SymLu** ∈ Init(GN) **alors** Analyser GN ; **si** SymLu ∈ Init(GV) **alors** Analyser GV ; **si** SymLu ≠ '.'**alors** Erreur **fsi** **sinon** Erreur **fsi** **sinon** Erreur **fsi**	**Plan phrase** : **Activer(Init(Phrase));** **Plan GN ;** **Plan GV ;** **Plan point** **Désactiver(Init(Phrase)) ;**

L'implantation du plan d'action Phrase en Delphi est immédiate :

```
procedure Plan_phrase;
begin
  Plan_GN;
  Plan_GV;
  Plan_point;
end;
```

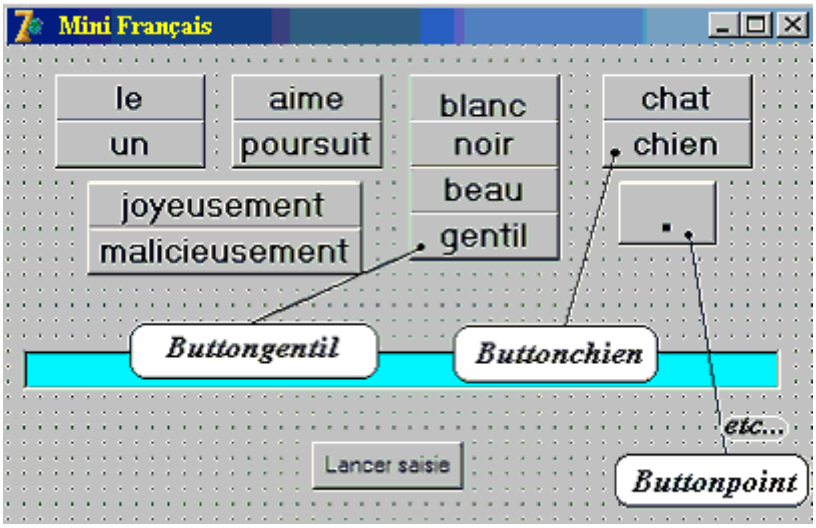
Deuxième plan d'action celui du symbole < LeNom >

Bloc Analyser LeNom : **si SymLu** ∈ Init(Adj) **alors** Analyser Adj; **si** SymLu ∈ Init(Nom) **alors** Analyser Nom **sinon** Erreur **fsi** **sinon** **si** SymLu ∈ Init(Nom) **alors** Analyser Nom; **si** SymLu ∈ Init(Adj) **alors** Analyser Adj; **sinon** Erreur **fsi** **sinon** Erreur **fsi** **fsi**	**Plan LeNom** **Activer(Init(LeNom));** **Saisie;** **Désactiver(Init(LeNom));** **si action** ∈**Init(Nom) alors** **Plan Nom ;** **Plan Adj** **sinon** **Plan Adj;** **Plan Nom** **fsi**

L'implantation du plan d'action LeNom en Delphi est immédiate :

```
procedure Plan_LeNom;
begin
  …. etc
end;
```

Code Delphi7 complet de l'exemple-2 *Interface de saisie du mini-français*




```
unit UFplanGF2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;
```

```
 type
  TFPPlanGF2 = class(TForm)
  ButtonLe: TButton;
  ButtonUn: TButton;
  Buttonblanc: TButton;
  Buttonnoir: TButton;
  Buttonbeau: TButton;
  Buttongentil: TButton;
  Buttonaime: TButton;
  Buttonpoursuit: TButton;
  Buttonchat: TButton;
  Buttonmalicieus: TButton;
  Buttonjoyeus: TButton;
  ButtonLancer: TButton;
  Edit_saisie: TEdit;
  Buttonchien: TButton;
  Buttonpoint: TButton;
  BitBtnFermer: TBitBtn;
  procedure ButtonLancerClick(Sender: TObject);
  procedure ButtonsClick(Sender: TObject);
  procedure FormCreate(Sender: TObject);
  procedure BitBtnFermerClick(Sender: TObject);
 private
 { Déclarations privées }
 public
 { Déclarations publiques }
  SymLu:string;
  ActionFaite , stop : boolean;
  procedure ActiverDesactiver(Elt:TButton; const etat:boolean);
  procedure InitGN(active:boolean);
  procedure InitLeNom(active:boolean);
  procedure InitGV(active:boolean);
  procedure InitSuite(active:boolean);
  procedure InitArt(active:boolean);
  procedure InitNom(active:boolean);
  procedure InitVerbe(active:boolean);
  procedure InitAdj(active:boolean);
  procedure InitAdv(active:boolean);
  procedure RAZTout;
  procedure AttenteSaisie;
  procedure PlanSuivant;
  procedure Plan_Art;
  procedure Plan_Nom;
  procedure Plan_Verbe;
  procedure Plan_Adj;
  procedure Plan_Adv;
  procedure Plan_LeNom;
  procedure Plan_GN;
  procedure Plan_Suite;
  procedure Plan_GV;
  procedure Plan_point;
  procedure Plan_phrase;
end;

 var
  FPPlanGF2: TFPPlanGF2;

implementation
```

```
{$R *.dfm }

procedure TFPPlanGF2.ActiverDesactiver(Elt:TButton;const etat:boolean);
begin
 Elt.enabled:=etat;
end;
//////////////////// LES INIT ////////////////////////////
procedure TFPPlanGF2.InitGN(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.ButtonLe,active);
 ActiverDesactiver(FPPlanGF2.ButtonUn,active)
end;

procedure TFPPlanGF2.InitLeNom(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonblanc,active);
 ActiverDesactiver(FPPlanGF2.Buttonnoir,active);
 ActiverDesactiver(FPPlanGF2.Buttonbeau,active);
 ActiverDesactiver(FPPlanGF2.Buttongentil,active);
 ActiverDesactiver(FPPlanGF2.Buttonchat,active);
 ActiverDesactiver(FPPlanGF2.Buttonchien,active)
end;

procedure TFPPlanGF2.InitGV(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonaime,active);
 ActiverDesactiver(FPPlanGF2.Buttonpoursuit,active)
end;

procedure TFPPlanGF2.InitSuite(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.ButtonLe,active);
 ActiverDesactiver(FPPlanGF2.ButtonUn,active);
 ActiverDesactiver(FPPlanGF2.Buttonjoyeus,active);
 ActiverDesactiver(FPPlanGF2.Buttonmalicieus,active)
end;

procedure InitArt(active:boolean);
begin
 InitGN(active)
end;

procedure TFPPlanGF2.InitNom(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonchat,active);
 ActiverDesactiver(FPPlanGF2.Buttonchien,active)
end;

procedure TFPPlanGF2.InitVerbe(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonaime,active);
 ActiverDesactiver(FPPlanGF2.Buttonpoursuit,active)
end;

procedure TFPPlanGF2.InitAdj(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonblanc,active);
 ActiverDesactiver(FPPlanGF2.Buttonnoir,active);
 ActiverDesactiver(FPPlanGF2.Buttonbeau,active);
 ActiverDesactiver(FPPlanGF2.Buttongentil,active)
```

```
end;

procedure TFPPlanGF2.InitAdv(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonjoyeus,active);
 ActiverDesactiver(FPPlanGF2.Buttonmalicieus,active)
end;
//////////////////// fin des INIT ///////////////////////
procedure TFPPlanGF2.RAZTout;
begin
 with FPPlanGF2 do
 begin
  InitArt(false);
  InitNom(false);
  InitVerbe(false);
  InitAdj(false);
  InitAdv(false);
  Buttonpoint.Enabled:=false;
  SymLu:='';
  ActionFaite:=false;
  Edit_saisie.text:='';
  stop:=false
 end
end;

procedure TFPPlanGF2.AttenteSaisie;
procedure Attendre;
begin
 if not ActionFaite then
 begin
  repeat
   Application.ProcessMessages
  until
  actionfaite ;
  ActionFaite:=false
 end
end{ Attendre};

procedure saisie(ch:string);
begin
 if ActionFaite=false then
  FPPlanGF2.Edit_saisie.text:= FPPlanGF2.Edit_saisie.text+' '+ch
 else
  actionfaite:=false
end{saisie};

begin
 Attendre;
 saisie(SymLu);
end{AttenteSaisie};

procedure TFPPlanGF2.PlanSuivant;
begin
 FPPlanGF2.ButtonLancer.enabled:=true;
end;
{--------------- Les procédures des plans d'actions ----------------}
procedure TFPPlanGF2.Plan_Art;
begin
 InitArt(true);
 AttenteSaisie;
```

```
 InitArt(false);
end;

procedure TFPPlanGF2.Plan_Nom;
begin
 InitNom(true);
 AttenteSaisie;
 InitNom(false);
end;

procedure TFPPlanGF2.Plan_Verbe;
begin
 InitVerbe(true);
 AttenteSaisie;
 InitVerbe(false);
end;

procedure TFPPlanGF2.Plan_Adj;
begin
 InitAdj(true);
 AttenteSaisie;
 InitAdj(false);
end;

procedure TFPPlanGF2.Plan_Adv;
begin
 InitAdv(true);
 AttenteSaisie;
 InitAdv(false);
end;

procedure TFPPlanGF2.Plan_LeNom;
begin
 InitLeNom(true);
 AttenteSaisie;
 InitLeNom(false);
 ActionFaite:=true; // action déjà saisie
 if (SymLu='chat') or (SymLu='chien') then
 begin
  Plan_Nom;
  Plan_Adj
 end
 else  // adjectif
 begin
  Plan_Adj;
  Plan_Nom
 end;
end;

procedure TFPPlanGF2.Plan_GN;
begin
 Plan_Art;
 Plan_LeNom;
end;

procedure TFPPlanGF2.Plan_Suite;
begin
 InitSuite(true);
 AttenteSaisie;
 InitSuite(false);
```

```
 ActionFaite:=true; // action déjà saisie
 if (SymLu='le') or (SymLu='un') then
 begin
  Plan_GN;
 end
 else  // adverbe
 begin
  Plan_Adv;
  Plan_GN
 end;
end;

procedure TFPPlanGF2.Plan_GV;
begin
 Plan_Verbe;
 Plan_Suite;
end;

procedure TFPPlanGF2.Plan_point;
begin
 FPPlanGF2.Buttonpoint.Enabled:=true;
 AttenteSaisie;
 FPPlanGF2.Buttonpoint.Enabled:=false;
end;

procedure TFPPlanGF2.Plan_phrase;
begin
 Plan_GN;
 Plan_GV;
 Plan_point;
end;

procedure TFPPlanGF2.ButtonLancerClick(Sender: TObject);
begin
 RAZTout;
 ButtonLancer.enabled:=false;
 Plan_phrase;
 PlanSuivant;
end;

procedure TFPPlanGF2.ButtonsClick(Sender: TObject);
begin
 ActionFaite:=true;
 if stop then
  halt;//l'utilisateur a demandé d'arrêter
 Symlu:=TButton(Sender).caption
end;

procedure TFPPlanGF2.FormCreate(Sender: TObject);
begin
 RAZTout;
end;

procedure TFPPlanGF2.BitBtnFermerClick(Sender: TObject);
begin
 stop:=true
end;

end.
```

Chapitre 7.4 Une projet d'IHM

Enquête fumeurs

Plan du chapitre:

1. Le projet de construction d'une borne interactive

1.1 Le marché avec le client
1.2 Aspect général d'un prototype
1.3 Partition de l'interface en zones d'action
1.4 Le mode attente utilisateur
1.5 Le mode consultation

2. Le mode saisie et les plans d'action

2.1 Graphe général des plans d'action
2.2 Graphe événementiel de la zone-1
2.3 Graphe événementiel de la zone-2
2.4 Graphe événementiel de la zone-3
2.5 Graphe événementiel de la zone-4
2.6 Graphe événementiel de la zone-5

3. Le reste du logiciel

3.1 Le menu lance la saisie du mot de passe

1. Le projet de construction d'une borne interactive

Programmons un exemple en utilisant les principes d'élaboration d'une interface par plans d'action.
Nous sommes l'I.N.S., un institut d'enquêtes et de sondages au service de clients qui nous commandent des enquêtes de données chiffrées que nous exploiterons ultérieurement. Le travail ci-dessous peut être réalisé en environ 500 lignes de Delphi. Il peut être réalisé par un seul programmeur (temps de programmation 50h environ) ou avec une équipe de trois étudiants à qui l'on confiera des activités différentes.

1.1 Le marché avec le client

Le client, une organisation de lutte contre le tabagisme, nous a commandé une enquête sur le public des fumeurs de cigarettes dont l'âge est compris entre 10 ans et 120 ans. Il désire obtenir un fichier de données recueillies auprès du public des deux sexes. Pour chaque personne interrogée nous devons stocker dans notre fichier :

- le sexe,
- l'âge actuel,
- depuis combien d'années la personne fume,
- le nombre de cigarettes fumées par jour.

Le client est conscient qu'il est difficile de demander à un individu le nombre exact de cigarettes fumées par jour. Il admet que nous proposions aux personnes interrogées une série de plages de consommations plutôt qu'un nombre précis.

Le client veut en même temps faire œuvre de pédagogie auprès des individus fumeurs sondés. Il souhaite que le sondé puisse se situer dans une fourchette de pourcentages de consommation sur toute la population des personnes déjà sondées. D'autre part le client dispose de tables de mortalité dont il a extrait des formules permettant d'évaluer le risque de cancer du poumon ou du larynx en fonction du sexe, de l'âge, de la durée du tabagisme et du nombre de cigarettes fumées par jour. Le client espère ainsi faire prendre conscience du risque accru d'apparition d'un cancer en cas de tabagisme prolongé.

1.2 Aspect général d'un prototype

Notre équipe de développement a réfléchi au problème et a décidé que nous construirions un prototype de borne interactive :

- Nous allons mettre en place un **logiciel ouvert** qui fonctionnera 24h sur 24, qui attendra le client et lui permettra de remplir son questionnaire " à la volée ".
- Le logiciel doit être verrouillé contre toute fausse manipulation de la part de l'utilisateur. L'équipe de développement a décidé d'employer la méthode de saisie dirigée par la syntaxe par plans d'action.
- L'utilisateur pourra revenir, pour correction, sur l'une quelconque des données qu'il aura entrées. Avant son stockage définitif dans le fichier, chaque transaction est mémorisée sur disque dès qu'elle a lieu.

- Le principe d'une sauvegarde générale journalière effectuée par une personne de l'INS a été retenu. Les actions de maintenance éventuelles effectuées par du personnel de l'INS s'effectueront à partir d'un menu spécial protégé par un mot de passe.

Voici le prototype d'interface proposé dans sa totalité :

1.3 Partition de l'interface en zones d'action

Actuellement tous les plans d'action sont visibles. Nous proposons de diviser l'interface en plusieurs zones d'action différentes.

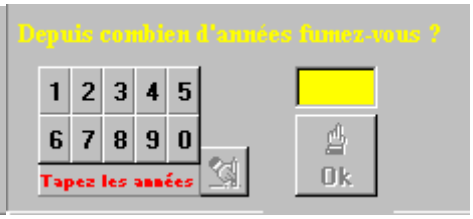
Cinq zones de saisies de l'information

- zone-1 d'indication du sexe du sondé,
- zone-2 de saisie de l'âge actuel,
- zone-3 de saisie de la durée du tabagisme,
- zone-4 de saisie du nombre de cigarettes par jour,
- zone-5 de correction des données entrées.

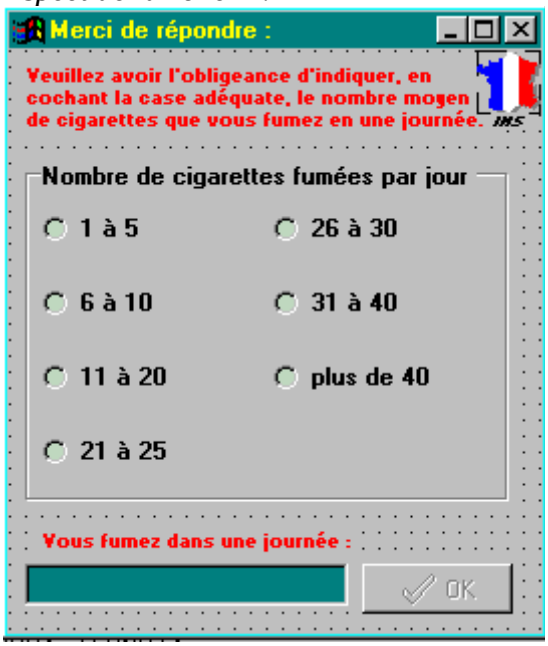
Aspect de la zone-1 :

Aspect de la zone-2 :

Aspect de la zone-3 :



Aspect de la zone-4 :

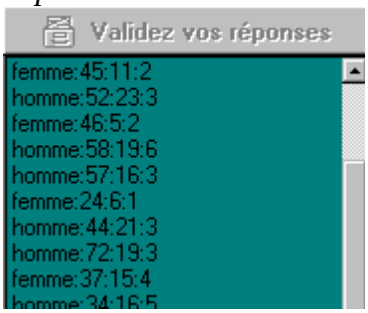


Aspect de la zone-5 :

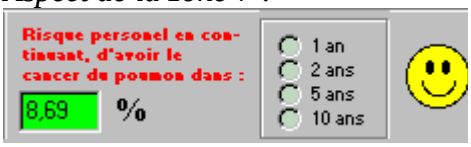
Deux zones de résultats consultables

- zone-6 de liste des données déjà entrées,
- zone-7 de consultation du risque de cancer

Aspect de la zone-6 :



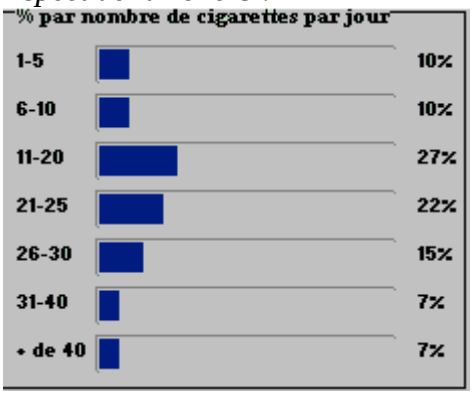
Aspect de la zone-7 :



Une zone d'affichage de résultats

zone-8 d'affichage des pourcentages de répartition

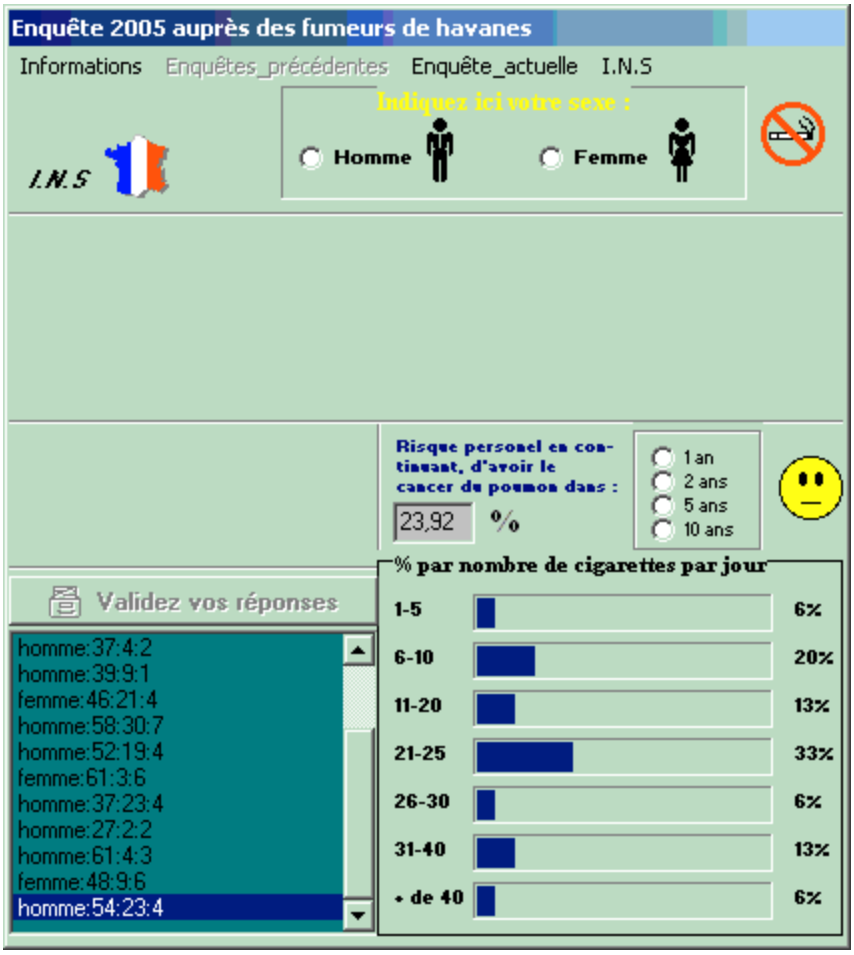
Aspect de la zone-8 :



1.4 Le mode attente utilisateur

Au lancement et après le passage de chaque sondé, l'interface est dans un état initial standard (écran d'accueil en mode attente). Elle peut prendre deux chemins d'action : soit le mode saisie (par séquencement : sondage), soit le mode consultation du fichier (visualisation de statistiques…).

Aspect visuel du mode de départ après passage de plusieurs sondés :



Remarque
un certain nombre de personnes ont déjà répondu à l'enquête. Les pourcentages apparaissent ainsi que la liste des réponses.
A ce stade, le sondé peut soit entrer ses données personnelles, et il entre dans le séquencement des plans d'action que nous allons voir ensuite, soit consulter les zones 6 et 7.

1.5 Le mode consultation

Le sondé peut clicker sur une donnée de la liste comme ci dessous. Les deux zones 5 et 7 sont immédiatement informées par les données sélectionnées dans la liste.

Graphe événementiel de la zone-6

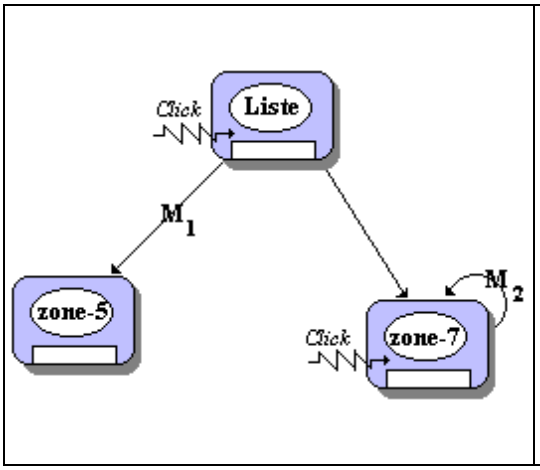
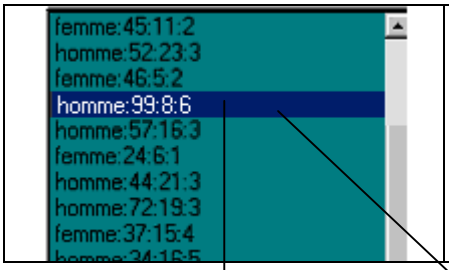


Table des états initiaux :

zone-5 désactivé
Liste activé
zone-7 activé
Homme désactivé
Femme désactivé
M1 = affichage dans zone désactivée (lecture seulement)
M2 = affichage des informations.

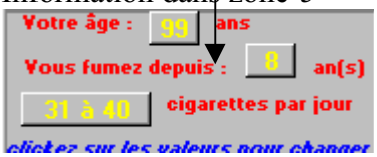
L'objet " liste " est implanté par l'objet visuel ListBoxReponses de la classe des TListBox.

Aspect visuel d'une consultation :

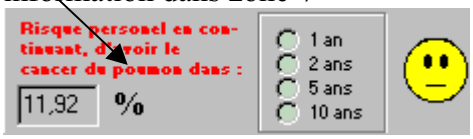


L'utilisateur sélectionne dans la liste une configuration (homme de 99 ans ayant fumé pendant 8 ans de 31 à 40 cigarettes par jour)

Information dans zone-5

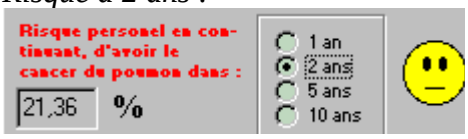


information dans zone-7

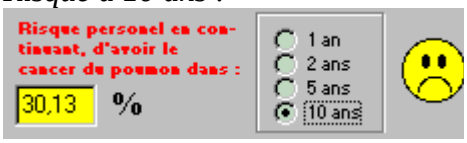


L'utilisateur peut alors consulter les différents pourcentages de risques associés à cette configuration en sélectionnant dans la zone-7 le risque à un an, à deux ans etc... L'affichage se fait visuellement d'une façon chiffrée et d'une façon imagée.

Risque à 2 ans :



Risque à 10 ans :

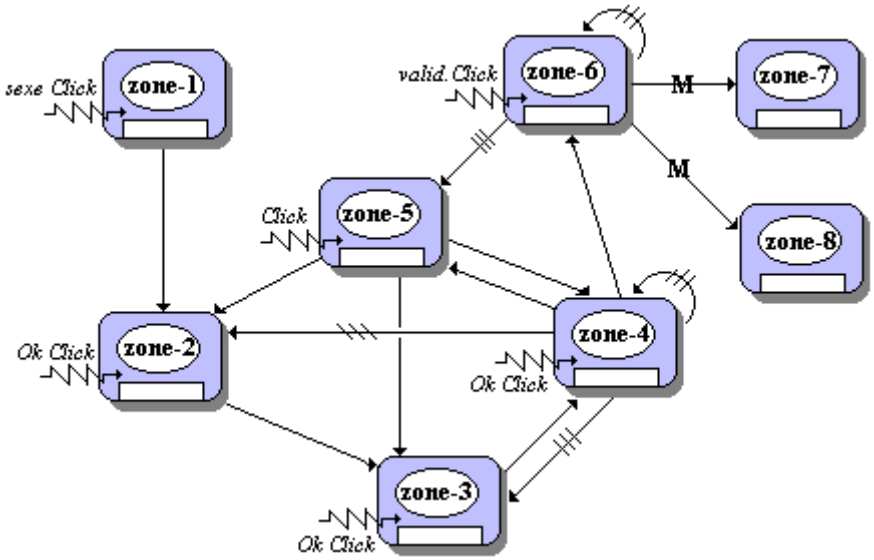


2. Le mode saisie et les plans d'action

Ce mode est dans le graphe événementiel le chemin produisant le plus grand nombre de lignes de code. C'est pourquoi il fait l'objet d'une étude à part.

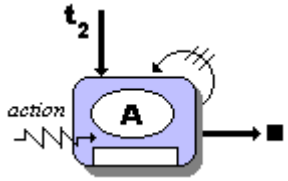
2.1 Graphe général des plans d'action

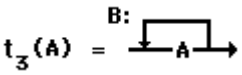
Nous présentons le graphe événementiel assurant le séquencement des plans d'action associés chacun à une des 8 zones décrites plus haut.



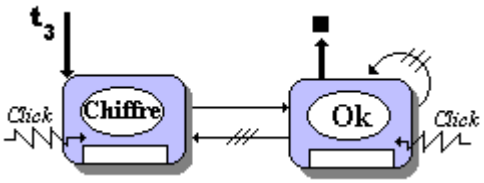
Nous avons choisi, pour la plupart des zones de représenter l'activation - désactivation par la propriété " visible ".
Pour les zones de saisie 2 et 3 où nous demandons à l'utilisateur d'entrer un nombre, nous avons choisi la saisie dirigée par la syntaxe. Un nombre est décrit par les diagrammes syntaxiques suivants :

Nombre



de la forme : 

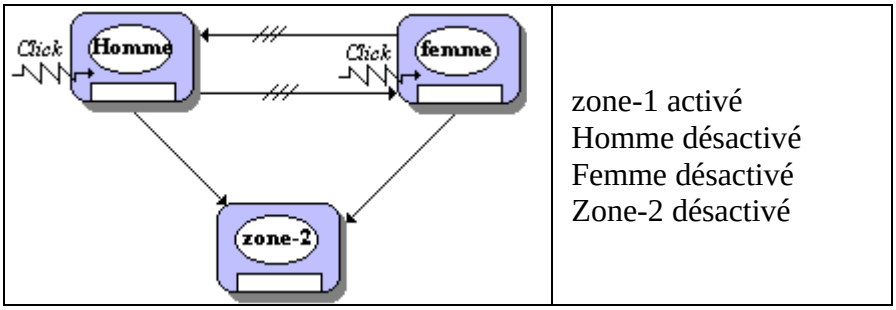
Opérateur t_3 dont nous avons déjà spécifié une implantation :



Chiffre sera implanté par un ensemble de boutons clickables associés chacun à un seul chiffre de 0 à 9.
Le bouton " Ok " permet de passer au plan d'action suivant.

2.2 Graphe événementiel de la zone-1

Table des états initiaux :



Les objets " homme " et " femme " sont implantés par les objets visuels de RadioButtonHomme et RadioButtonFemme de la classe des TRadioButton.

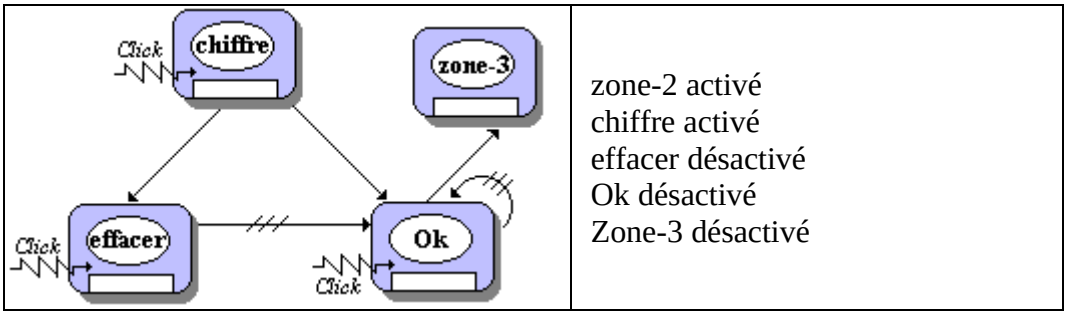
L'activation de la zone-2 (plan suivant)consiste à la rendre visible.

L'action du click sur l'une des deux cases dans zone-1

fait passer au plan d'action suivant.

2.3 Graphe événementiel de la zone-2

Table des états initiaux :



Les 10 objets " chiffre " sont implantés par les objets visuels de SpeedButtonAge1 à SpeedButtonAge10, les objets " effacer " et " ok " sont des objets visuels de type boutons poussoirs. L'activation de la zone-3 (plan suivant)consiste à la rendre visible.

Aspect visuel du deuxième plan d'action :

zone-2 avant saisie

à

zone-2 pendant saisie

le click sur le bouton Ok fait passer au plan d'action suivant

2.4 Graphe événementiel de la zone-3

Strictement identique au précédent avec comme seule différence le fait que l'objet (bouton poussoir) " ok " active la zone-4.

Aspect visuel du troisième plan d'action :

zone-3 avant saisie

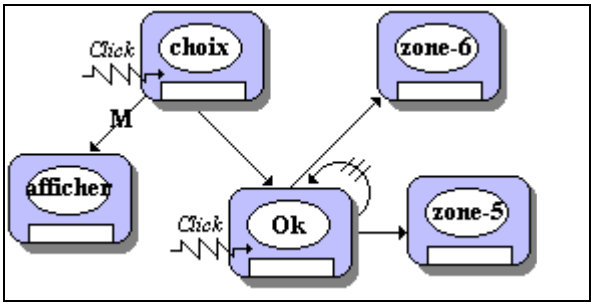
à

zone-3 pendant saisie

le click sur le bouton Ok fait passer au plan d'action suivant.

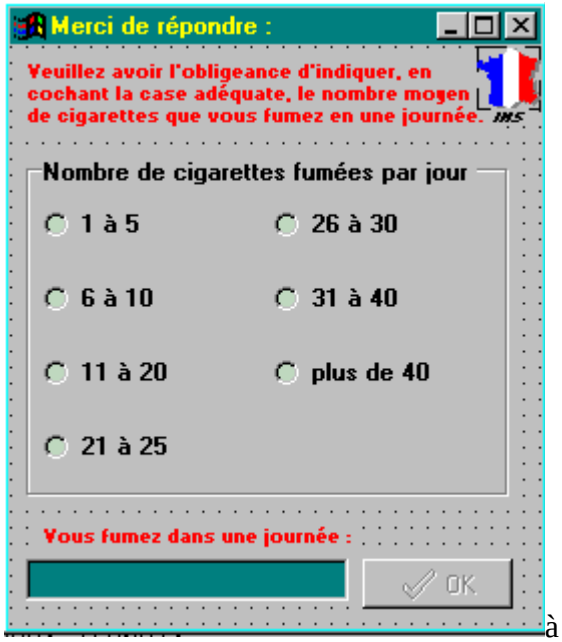
2.5 Graphe événementiel de la zone-4

Table des états initiaux :

	zone-4 activé Ok désactivé afficher désactivé Zone-5 désactivé Zone-6 désactivé

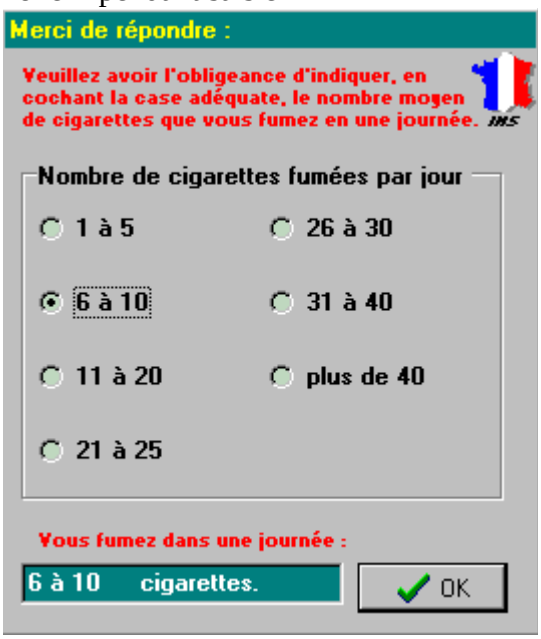
Aspect visuel du quatrième plan d'action :

zone-4 avant saisie



à

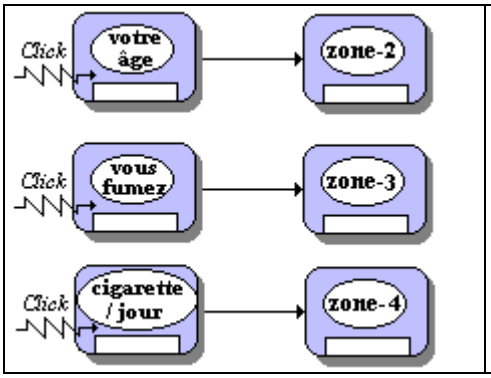
zone-4 pendant saisie



le click sur le bouton Ok fait passer au plan d'action suivant. **¯**

2.6 Graphe événementiel de la zone-5

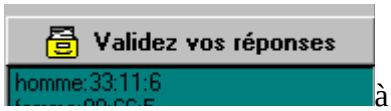
Table des états initiaux :



zone-6 activé
votre âge activé
vous fumez activé
cigarette/j activé
Zone-2 désactivé
Zone-3 désactivé
Zone-4 désactivé

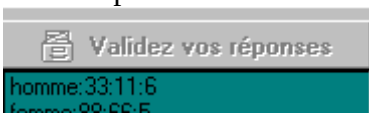
Aspect visuel du cinquième plan d'action :
Le bouton de validation de la zone-6 est activé.

zone-6 avant saisie



à

zone-6 après saisie

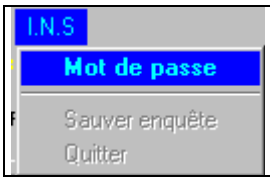


Fin du séquencement des plans d'action : l'interface est remise à l'état initial.

3. Le reste du logiciel

Ce qui reste à décrire dans notre logiciel figure à des fins pédagogiques afin que le lecteur puisse voir que la notion de séquencement de plan d'action ne se limite pas à l'utilisation des seuls objets visuels que sont les boutons. La liaison est faite entre une autre fiche de dialogue permettant de saisir un mot de passe et l'activation-désactivation de zones de menu.

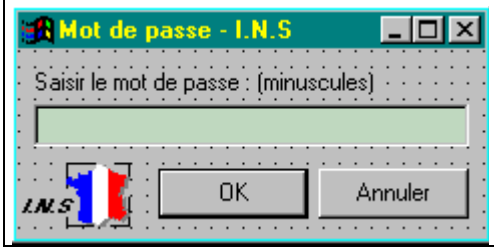
3.1 Le menu lance la saisie du mot de passe



Remarque
L'utilisateur clique sur la zone surlignée.

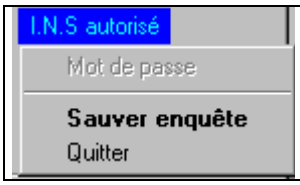
Le clic appelle le gestionnaire suivant :

```
procedure TForm1.Motdepasse1Click(Sender: TObject);
{le menu affiche une fiche de saisie du mot de passe dans UFmotPasse}
begin
  MotdePass.Showmodal;
end;
```



La fiche ci-contre (name = MotdePass) qui est dans l'unité *UFmotPasse*, est affichée en catégorie modale (impossible de cliquer ailleurs).

Si le mot de passe est valide, la fiche " MotdePass " se ferme et informe le menu " INS " en désactivant une zone (la zone mot de passe) et en activant les deux qui étaient inactivées au départ.



Remarque
le séquencement d'action est assuré aussi de cette façon.

Chapitre 7.5 utilisation des bases de données

Plan du chapitre:

1. Introduction et Généralités

2. Le modèle de données relationnel

3. Principes fondamentaux d'une algèbre relationnelle

4. SQL et Algèbre relationnelle

5. Exemple de communication entre Delphi et les BD

1. Introduction et Généralités

1.1 Notion de système d'information

L'informatique est une science du traitement de l'information, laquelle est représentée par des données Aussi, très tôt, on s'est intéressé aux diverses manières de pouvoir stocker des données dans des mémoires auxiliaires autres que la mémoire centrale. Les données sont stockées dans des périphériques dont les supports physiques ont évolué dans le temps : entre autres, d'abord des cartes perforées, des bandes magnétiques, des cartes magnétiques, des mémoires à bulles magnétiques, puis aujourd'hui des disques magnétiques, ou des CD-ROM ou des DVD.

La notion de fichier est apparue en premier : le fichier regroupe tout d'abord des objets de même nature , des enregistrements. Pour rendre facilement exploitables les données d'un fichier, on a pensé à différentes méthodes d'accès (accès séquentiel, direct, indexé).

Toute application qui gère des systèmes physiques doit disposer de paramètres sémantiques décrivant ces systèmes afin de pouvoir en faire des traitements. Dans des systèmes de gestion de clients les paramètres sont très nombreux (noms, prénoms, adresse, n°Sécu, sport favori, est satisfait ou pas,..) et divers (alphabétiques, numériques, booléens, ...).

Dès que la quantité de données est très importante, les fichiers montrent leurs limites et il a fallu trouver un moyen de stocker ces données et de les organiser d'une manière qui soit facilement accessible.

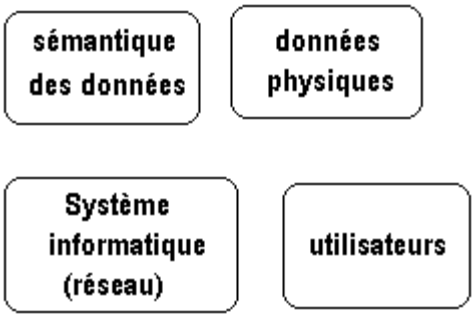
Base de données (BD)

> Une BD est composée de données stockées dans des mémoires de masse sous une forme structurée, et accessibles par des applications différentes et des utilisateurs différents. Une BD doit pouvoir être utilisée par plusieurs utilisateurs en "même temps".

Une base de données est structurée par définition, mais sa structuration doit avoir un caractère universel : il ne faut pas que cette structure soit adaptée à une application particulière, mais qu'elle puisse être utilisable par plusieurs applications distinctes. En effet, un même ensemble de données peut être commun à plusieurs systèmes de traitement dans un problème physique (par exemple la liste des passagers d'un avion, stockée dans une base de données, peut aussi servir au service de police à vérifier l'identité des personnes interdites de séjour, et au service des douanes pour associer des bagages aux personnes….).

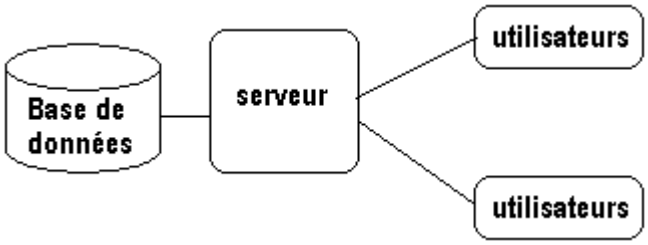
Système d'information

> Dans une entreprise ou une administration, la structure sémantique des données, leur organisation logique et physique, le partage et l'accès à de grandes quantités de données grâce à un système informatique, se nomme un système d'information.



Les entités qui composent un système d'information

L'organisation d'un SI relève plus de la gestion que de l'informatique et n'a pas exactement sa place dans un document sur la programmation. En revanche la cheville ouvrière d'un système d'information est un outil informatique appelé un **SGBD** (système de gestion de base de données) qui repose essentiellement sur un système informatique composé traditionnellement d'une **BD** et d'un réseau de postes de travail consultant ou mettant à jour les informations contenues dans la base de données, elle-même généralement située sur un ordinateur-serveur.



Système de Gestion de Base de Données (SGBD)

Un SGBD est un ensemble de logiciels chargés d'assurer les fonctions minimales suivantes :

- Le maintien de la cohérence des données entre elles,
- le contrôle d'intégrité des données accédées,
- les autorisations d'accès aux données,
- les opérations classiques sur les données (consultation, insertion , modification, suppression)

La cohérence des données est subordonnée à la définition de contraintes d'intégrité qui sont des règles que doivent satisfaire les données pour être acceptées dans la base. Les contraintes d'intégrité sont contrôlées par le moteur du SGBD :

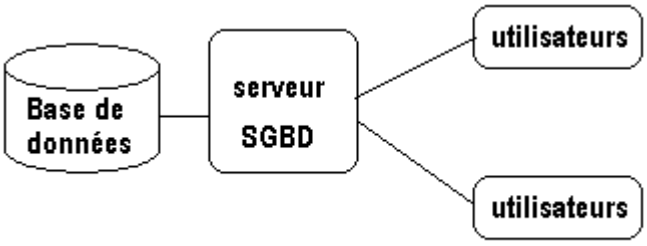
- au niveau de chaque champ, par exemple le : prix est un nombre positif, la date de naissance est obligatoire.
- Au niveau de chaque table - voir plus loin la notion de clef primaire : deux personnes ne doivent pas avoir à la fois le même nom et le même prénom.
- Au niveau des relations entre les tables : contraintes d'intégrité référentielles.

Par contre la redondance des données (formes normales) **n'est absolument pas vérifiée automatiquement** par les SGBD, il faut faire des requêtes spécifiques de recherche

d'anomalies (dites post-mortem) **à postériori**, ce qui semble être une grosse lacune de ces systèmes puisque les erreurs sont déjà présentes dans la base !

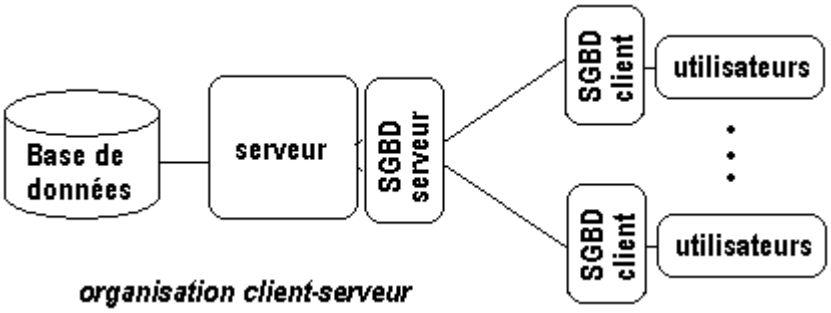
On organise actuellement les SGBD selon deux modes :

L'organisation locale selon laquelle le SGBD réside sur la machine où se trouve la base de données :



organisation locale

L'organisation client-serveur selon laquelle sur le SGBD est réparti entre la machine serveur locale supportant la BD (partie SGBD serveur) et les machines des utilisateurs (partie SGBD client). Ce sont ces deux parties du SGBD qui communiquent entre elles pour assurer les transactions de données :



organisation client-serveur

Le caractère généraliste de la structuration des données induit une description abstraite de l'objet BD (Base de données). Les applications étant indépendantes des données, ces dernières peuvent donc être manipulées et changées indépendamment du programme qui y accédera en implantant les méthodes générales d'accès aux données de la base, conformément à sa structuration abstraite.

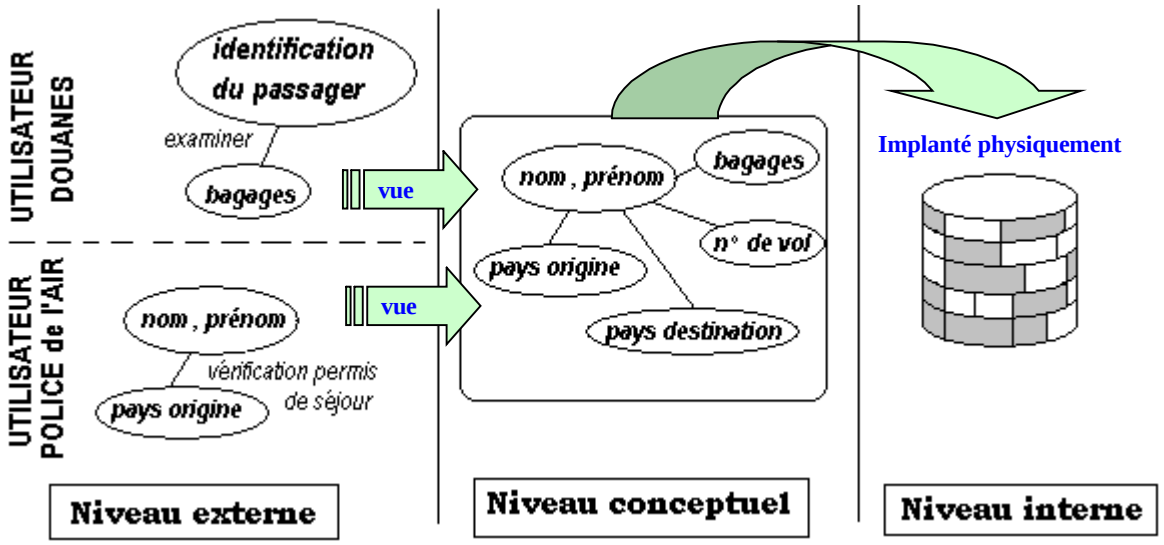
Une Base de Données peut être décrite de plusieurs points de vue, selon que l'on se place du côté de l'utilisateur ou bien du côté du stockage dans le disque dur du serveur ou encore du concepteur de la base.

Il est admis de nos jours qu'une **BD** est décrite en trois niveaux d'abstraction : un seul niveau a une existence matérielle physique et les deux autres niveaux sont une explication abstraite de ce niveau matériel.

Les 3 niveaux d'abstraction définis par l'ANSI depuis 1975

- **Niveau externe** : correspond à ce que l'on appelle une vue de la BD ou la façon dont sont perçues au niveau de l'utilisateur les données manipulées par une certaine application (vue abstraite sous forme de schémas)
- **Niveau conceptuel** : correspond à la description abstraite des composants et des processus entrant dans la mise en œuvre de la BD. Le niveau conceptuel est le plus important car il est le résultat de la traduction de la description du monde réel à l'aide d'expressions et de schémas conformes à un modèle de définition des données.
- **Niveau interne** : correspond à la description informatique du stockage physique des données (fichiers séquentiels, indexages, tables de hachage,…) sur le disque dur.

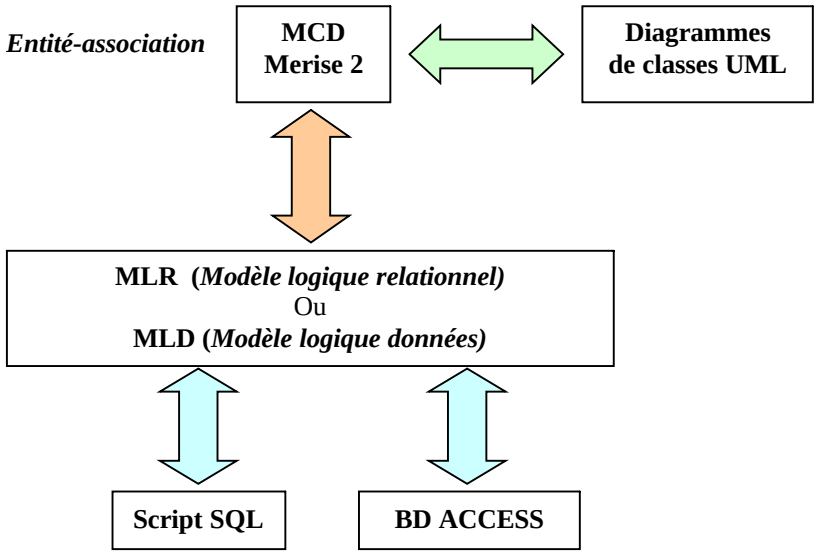
Figurons pour l'exemple des passagers d'un avion , stockés dans une base de données de la compagnie aérienne, sachant qu'en plus du personnel de la compagnie qui a une vue externe commerciale sur les passagers, le service des douanes peut accéder à un passager et à ses bagages et la police de l'air peut accéder à un passager et à son pays d"embarquement.



Le niveau conceptuel forme l'élément essentiel d'une BD et donc d'un SGBD chargé de gérer une BD, il est décrit avec un modèle de conception de données MCD avec la méthode française Merise qui est très largement répandu, ou bien par le formalisme des diagrammes de classes UML qui prend une part de plus en plus grande dans le formalisme de description conceptuelle des données (rappelons qu'UML est un langage de modélisation formelle, orienté objet et graphique ; Merise2 a intégré dans Merise ces concepts mais ne semble pas beaucoup être utilisé). Nous renvoyons le lecteur intéressé par cette partie aux très nombreux ouvrages écrits sur Merise ou sur UML.

Dans la pratique actuelle les logiciels de conception de BD intègrent à la fois la méthode Merise 2 et les diagrammes de classes UML. Ceci leur permet surtout la génération automatique et semi-automatique (paramétrable) de la BD à partir du modèle conceptuel sous forme de scripts (programmes simples) SQL adaptés aux différents SGBD du marché (ORACLE, SYBASE, MS-SQLSERVER,…) et les différentes versions de la BD ACCESS. Les logiciels de conception actuels permettent aussi la rétro-génération (ou reverse engeneering) du modèle à partir d'une BD existante, cette fonctionnalité est très utile pour reprendre un travail mal documenté.

En résumé pratique :



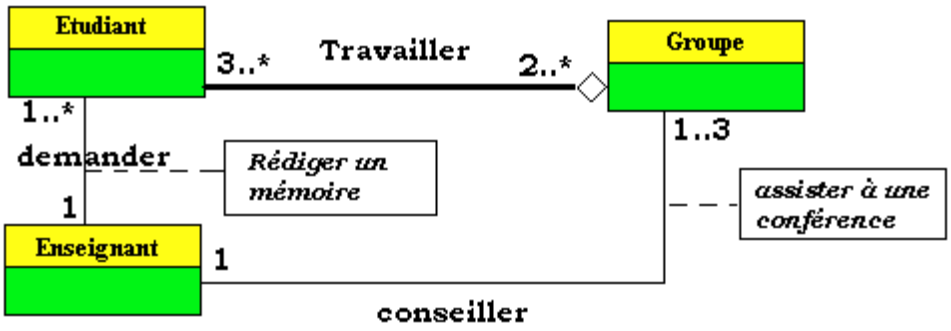
C'est en particulier le cas du logiciel français WIN-DESIGN dont une version démo est disponible à www.win-design.com et de son rival POWER-AMC (ex AMC-DESIGNOR).

Signalons enfin un petit logiciel plus modeste, très intéressant pour débuter avec version limitée seulement par la taille de l'exemple : CASE-STUDIO chez CHARONWARE. Les logiciels basés uniquement sur UML sont , à ce jour, essentiellement destinés à la génération de code source (Java, Delphi, VB, C++,…), les versions **Community** (versions logicielles libres) de ces logiciels ne permettent pas la génération de BD ni celle de scripts SQL.

Les quelques schémas qui illustreront ce chapitre seront décrits avec le langage UML.

L'exemple ci-après schématise en UML le mini-monde universitaire réel suivant :

- un enseignant pilote entre 1 et 3 groupes d'étudiants,
- un enseignant demande à 1 ou plusieurs étudiants de rédiger un mémoire,
- un enseignant peut conseiller aux groupes qu'il pilote d'aller assister à une conférence,
- un groupe est constitué d'au moins 3 étudiants,
- un étudiant doit s'inscrire à au moins 2 groupes.



Si le niveau conceptuel d'une BD est assis sur un modèle de conceptualisation de haut niveau (Merise, UML) des données, il est ensuite fondamentalement traduit dans le Modèle Logique de représentation des Données (MLD). Ce dernier s'implémentera selon un modèle physique des données.

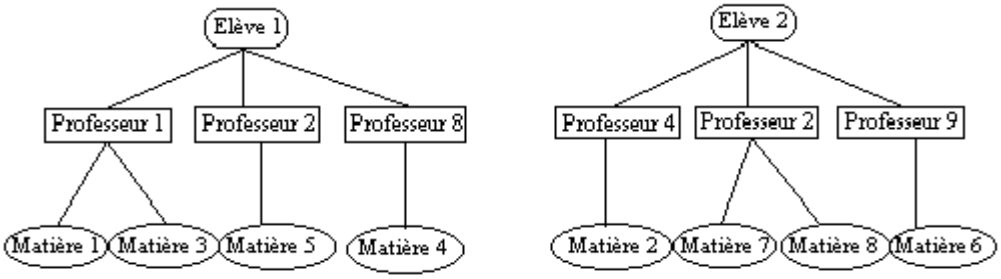
Il existe plusieurs MLD **M**odèles **L**ogiques de **D**onnées et plusieurs modèles physiques, et pour un même MLD, on peut choisir entre plusieurs modèles physiques différents.

Il existe 5 grands modèles logiques pour décrire les bases de données.

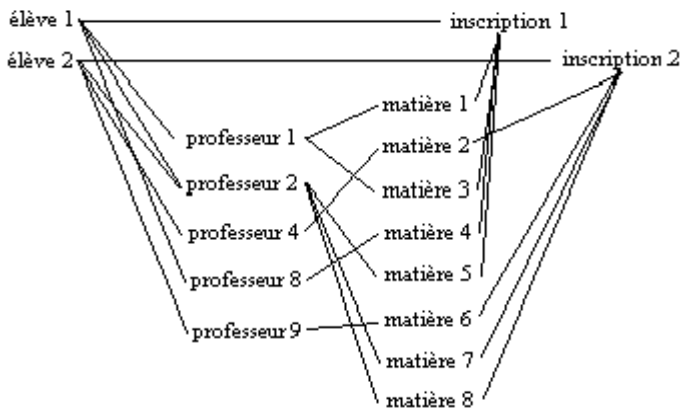
Les modèles de données historiques

(Prenons un exemple comparatif où des élèves ont des cours donnés par des professeurs leur enseignant certaines matières (les enseignants étant pluridisciplinaires)

- **Le modèle hiérarchique:** l'information est organisée de manière arborescente, accessible uniquement à partir de la racine de l'arbre hiérarchique. Le problème est que les points d'accès à l'information sont trop restreints.



- **Le modèle réseau:** toutes les informations peuvent être associées les unes aux autres et servir de point d'accès. Le problème est la trop grande complexité d'une telle organisation.



- **Le modèle relationnel**: toutes les relations entre les objets contenant les informations sont décrites et représentées sous la forme de tableaux à 2 dimensions.

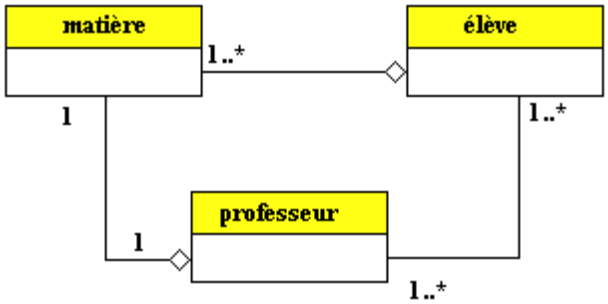
élève 1	matière 1
élève 1	matière 3
élève 1	matière 4
élève 1	matière 5
élève 2	matière 2
élève 2	matière 6
élève 2	matière 7
élève 2	matière 8

professeur 1	matière 1
professeur 1	matière 3
professeur 2	matière 5
professeur 2	matière 7
professeur 2	matière 8
professeur 4	matière 2
professeur 8	matière 4
professeur 9	matière 6

Dans ce modèle, la gestion des données (insertion, extraction,…) fonctionne selon la théorie mathématique de l'algèbre relationnelle. C'est le modèle qui allie une grande indépendance vis à vis des données à une simplicité de description.

- **Le modèle par déduction :** comme dans le modèle relationnel les données sont décrites et représentées sous la forme de tableaux à 2 dimensions. La gestion des données (insertion, extraction,…) fonctionne selon la théorie mathématique du calcul dans la logique des prédicats. Il ne semble exister de SGBD commercial directement basé sur ce concept. Mais il est possible de considérer un programme Prolog (programmation en logique) comme une base de données car il intègre une description des données. Ce sont plutôt les logiciels de réseaux sémantiques qui sont concernés par cette approche (cf. logiciel AXON).

- **Le modèle objet :** les données sont décrites comme des classes et représentées sous forme d'objets, un modèle **relationnel-objet** devrait à terme devenir le modèle de base.



L'expérience montre que le modèle relationnel s'est imposé parce qu'il était le plus simple en terme d'indépendance des données par rapport aux applications et de facilité de représenter les données dans notre esprit. C'est celui que nous décrirons succinctement dans la suite de ce chapitre.

2.Le modèle de données relationnel

Défini par EF Codd de la société IBM dès 1970, ce modèle a été amélioré et rendu opérationnel dans les années 80 sous la forme de SBGD-R (SGBD Relationnels). Ci-dessous une liste non exhaustive de tels SGBD-R :

Access de Microsoft,
Oracle,
DB2 d'IBM,
Interbase de Borland,
SQL server de microsoft,
Informix,
Sybase,
MySQL,
PostgreSQL, ….

Nous avons déjà vu dans un précédent chapitre, la notion de relation binaire : une relation binaire R est un sous-ensemble d'un produit cartésien de deux ensembles finis E et F que nous nommerons domaines de la relation R :

$$R \subseteq E \times F$$

Cette définition est généralisable à n domaines, nous dirons que **R** est une relation n-aire sur les domaines E_1 , E_2 , … , E_n si et seulement si :

$$R \subseteq E_1 \times E_2 \ldots \times E_n$$

Les ensembles E_k peuvent être définis comme en mathématiques : en extension ou en compréhension :

$E_k = \{ 12 , 58 , 36 , 47 \}$ *en extension*

$E_k = \{ x / (x \text{ est entier}) \textbf{ et(} x \in [1, 20]) \}$ *en compréhension*

Notation

si nous avons: $R = \{ (v_1 , v_2 \ldots , v_n) \}$,
Au lieu d'écrire : $(v_1 , v_2 \ldots , v_n) \in R$, on écrira $R(v_1 , v_2 , \ldots , v_n)$

Exemple de déclarations de relations :

Passager (nom, prénom, n° de vol, nombre de bagages) , cette relation contient les informations utiles sur un passager d'une ligne aérienne.

Personne (nom, prénom) , cette relation caractérise une personne avec deux attributs

Enseignement (professeur, matière) , cette relation caractérise un enseignement avec le nom de la matière et le professeur qui l'enseigne.

Schéma d'une relation

On appelle schéma de la relation R : $R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$
Où $(a_1 , a_2 \ldots , a_n)$ sont appelés les **attributs**, chaque attribut a_k indique comment est utilisé dans la relation R le domaine E_k , chaque attribut prend sa valeur dans le domaine qu'il définit, nous notons val(a_k)= v_k où v_k est un élément (une valeur) quelconque de l'ensemble E_k (domaine de l'attribut a_k).

Convention : lorsqu'il n'y a pas de valeur associée à un attribut dans un n-uplet, on convient de lui mettre une valeur spéciale notée **null**, indiquant l'absence de valeur de l'attribut dans ce n-uplet.

Degré d'une relation

On appelle degré d'une relation, le nombre d'attributs de la relation.

Exemple de schémas de relations :

Passager (nom : chaîne, prénom : chaîne, n° de vol : entier, nombre de bagages : entier) relation de degré 4.

Personne (nom : chaîne, prénom : chaîne) relation de degré 2.

Enseignement (professeur : ListeProf, matière : ListeMat) relation de degré 2.

*Attributs : prenons le schéma de la relation **Enseignement***

Enseignement (professeur : ListeProf, matière : ListeMat). C'est une relation binaire (degré 2) sur les deux domaines ListeProf et ListeMat. L'attribut professeur joue le rôle d'un paramètre formel et le domaine ListeProf celui du type du paramètre.

Supposons que :

ListeProf ={ Poincaré, Einstein, Lavoisier, Raimbault , Planck }
ListeMat = { mathématiques, poésie , chimie , physique }

L'attribut professeur peut prendre toutes valeurs de l'ensemble ListeProf :

Val(professeur) = Poincaré, …. , Val(professeur) = Raimbault

Si l'on veut dire que le poste d'enseignant de chimie n'est pas pourvu on écrira :

Le couple (**null** , chimie) est un couple de la relation **Enseignement.**

Enregistrement dans une relation

Un n-uplet $(val(a_1), val(a_2) \ldots , val(a_n)) \in R$ est appelé un enregistrement de la relation R. Un enregistrement est donc constitué de valeurs d'attributs.

Dans l'exemple précédent (Poincaré , mathématiques), (Raimbault , poésie) , (**null** , chimie) sont trois enregistrements de la relation **Enseignement**.

Clef d'une relation

Si l'on peut caractériser d'une façon **bijective** tout n-uplet d'attributs $(a_1 , a_2 \ldots , a_n)$ avec seulement un sous-ensemble restreint $(a_{k1} , a_{k2} \ldots , a_{kp})$ avec $p < n$, de ces attributs, alors ce sous-ensemble est appelé une **clef** de la relation. Une relation peut avoir plusieurs clefs, nous choisissons l'une d'elle en la désignant comme **clef primaire de la relation**.

Clef minimale d'une relation

On a intérêt à ce que la clef **primaire soit minimale** en nombre d'attributs, car il est clair que si un sous-ensemble à p attributs $(a_{k1} , a_{k2} \ldots , a_{kp})$ est une clef, tout sous-ensemble à p+m attributs dont les p premiers sont les $(a_{k1} , a_{k2} \ldots , a_{kp})$ est aussi une clef :
$(a_{k1} , a_{k2} \ldots , a_{kp} , a_0 ,a_1)$
$(a_{k1} , a_{k2} \ldots , a_{kp} , a_{10} , a_5 , a_9 , a_2)$ sont aussi des clefs etc…

Il n'existe aucun moyen méthodique formel général pour trouver une clef primaire d'une relation, il faut observer attentivement. Par exemple :

- Le code Insee est une clef primaire permettant d'identifier les personnes.
- Si le couple (nom, prénom) peut suffire pour identifier un élève dans une classe d'élèves, et pourrait être choisi comme clef primaire, il est insuffisant au niveau d'un lyçée où il est possible que l'on trouve plusieurs élèves portant le même nom et le même premier prénom ex: (martin, jean).

Convention : on souligne dans l'écriture d'une relation dont on a déterminé une clef primaire, les attributs faisant partie de la clef.

Clef secondaire d'une relation

Tout autre clef de la relation qu'une clef primaire (minimale) , exemple :
Si $(a_{k1} , a_{k2} \ldots , a_{kp})$ est un clef primaire de R
$(a_{k1} , a_{k2} \ldots , a_{kp} , a_0 ,a_1)$ et $(a_{k1} , a_{k2} \ldots , a_{kp} , a_{10} , a_5 , a_9 , a_2)$ sont des clefs secondaires.

Clef étrangère d'une relation

Soit $(a_{k1} , a_{k2} \ldots , a_{kp})$ un p-uplet d'attributs d'une relation R de degré n. [$R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$]

Si $(a_{k1} , a_{k2} \ldots , a_{kp})$ est une clef primaire d'une autre relation Q on dira que $(a_{k1} , a_{k2} \ldots , a_{kp})$ est une clef étrangère de R.

Convention : on met un # après chaque attribut d'une clef étrangère.

Exemple de clef secondaire et clef étrangère :

Passager (nom# : chaîne, prénom# : chaîne , n° de vol : entier, nombre de bagages : entier, n° client : entier) relation de degré 5.

Personne (nom : chaîne, prénom : chaîne , âge : entier, civilité : Etatcivil) relation de degré 4.

n° client est une clef primaire de la relation **Passager**.
(nom, n° client) est une clef secondaire de la relation **Passager**.
(nom, n° client , n° de vol) est une clef secondaire de la relation **Passager**….etc

(nom , prénom) est une clef primaire de la relation **Personne**, comme (nom# , prénom#) est aussi un couple d'attributs de la relation **Passager**, c'est une clef étrangère de la relation **Passager**.

On dit aussi que dans la relation **Passager**, le couple d'attributs (nom# , prénom#) réfère à la relation **Personne**.

Règle d'intégrité référentielle

Toutes les valeurs d'une clef étrangère (v_{k1} , v_{k2} … , v_{kp}) se retrouvent comme valeur de la clef primaire de la relation référée (ensemble des valeurs de la clef étrangère est **inclus** au sens large dans l'ensemble des valeurs de la clef primaire)

Reprenons l'exemple précédent

(nom , prénom) est une clef étrangère de la relation **Passager**, c'est donc une clef primaire de la relation **Personne** : les domaines (liste des noms et liste des prénoms associés au nom doivent être strictement les mêmes dans **Passager** et dans **Personne**, nous dirons que la clef étrangère respecte la contrainte d'intégrité référentielle.

Règle d'intégrité d'entité

Aucun des attributs participant à une clef primaire ne peut avoir la valeur **null**.

Nous définirons la valeur **null**, comme étant une valeur spéciale n'appartenant pas à un domaine spécifique mais ajoutée par convention à tous les domaines pour indiquer qu'un champ n'est pas renseigné.

Représentation sous forme tabulaire

Reprenons les relations Passager et Personne et figurons un exemple pratique de valeurs des relations.

Passager (nom# : chaîne, prénom# : chaîne, n° de vol : entier, nombre de bagages : entier, n° client : entier).

Personne (nom : chaîne, prénom : chaîne, âge : entier, civilité : Etatcivil) relation de degré 4.

Nous figurons les tables de valeurs des deux relations

Clef étrangère

Passager :

nom#	prénom#	N°vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45622	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Einstein	Albert	75906	**null**	858547
Lavoisier	Antoine	45644	4	235002

Clef primaire

Clef primaire

Personne:

nom	prénom	âge	Civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Nous remarquons que la compagnie aérienne attribue un numéro de client unique à chaque personne, c'est donc un bon choix pour une clef primaire.

Les deux tables (relations) ont deux colonnes qui portent les mêmes noms colonne **nom** et colonne **prénom**, ces deux colonnes forment une clef primaire de la table **Personne**, c'est donc une clef étrangère de **Passager** qui réfère **Personne.**

En outre, cette clef étrangère respecte la contrainte d'intégrité référentielle : la liste des valeurs de la clef étrangère dans **Passager** est incluse dans la liste des valeurs de la clef primaire associée dans **Personne**.

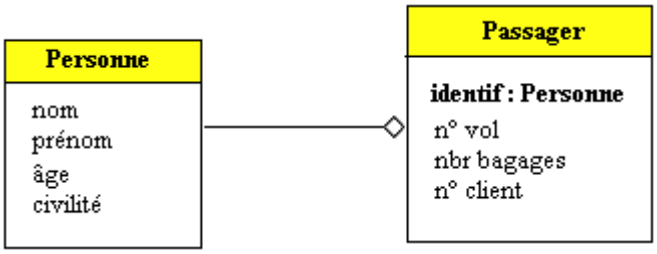


Diagramme UML modélisant la liaison Passager-Personne

Ne pas penser qu'il en est toujours ainsi, par exemple voici une autre relation **Passager2** dont la clef étrangère ne respecte pas la contrainte d'intégrité référentielle :

Clef étrangère , réfère Personne

Passager2 :

nom	prénom	N°vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Einstein	Albert	75906	**null**	858547
Picasso	Pablo	12896	5	458023

En effet, le couple (Picasso , Pablo) n'est pas une valeur de la clef primaire dans la table **Personne**.

Principales règles de normalisation d'une relation

1ère forme normale :

Une relation est dite en première forme normale si, chaque attribut est représenté par un identifiant unique (les valeurs ne sont pas des ensembles, des listes,…) .Ci-dessous une relation qui n'est pas en 1ère forme normale car l'attribut **n° vol** est multivalué (il peut prendre 2 valeurs) :

nom	prénom	N°vol	Nbr bagage	N° client
Einstein	Albert	45622 , 75906	2	154565
Lavoisier	Antoine	45644 , 45622	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Picasso	Pablo	12896	5	458023

En pratique, il est très difficile de faire vérifier automatiquement cette règle, dans l'exemple proposé on pourrait imposer de passer par un masque de saisie afin que le N°vol ne comporte que 5 chiffres.

2ème forme normale :

Une relation est dite en deuxième forme normale si, elle est **en première forme normale** et si chaque attribut qui n'est pas une clef, dépend entièrement de tous les attributs qui composent la clef primaire. La relation **Personne** (nom : chaîne, prénom : chaîne , age : entier , civilité : Etatcivil) est en deuxième forme normale :

nom	**prénom**	**âge**	**Civilité**
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut âge ne dépend que du nom et du prénom, de même pour l'attribut civilité.

La relation **Personne3** (nom : chaîne, prénom : chaîne , age : entier , civilité : Etatcivil) qui a pour clef primaire (nom , âge) n'est pas en deuxième forme normale :

nom	**prénom**	**âge**	**Civilité**
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut Civilité ne dépend que du nom et non pas de l'âge ! Il en est de même pour le prénom, soit il faut changer de clef primaire et prendre (nom, prénom) soit si l'on conserve la clef primaire (nom , âge) , il faut décomposer la relation **Personne3** en deux autres relations **Personne31** et **Personne32** :

nom	**âge**	**Civilité**
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Raimbault	25	célibataire
Poincaré	45	marié

Personne31(nom : chaîne, age : entier , civilité : Etatcivil) :
2ème forme normale

nom	**âge**	**prénom**
Einstein	45	Albert
Lavoisier	41	Antoine
Planck	52	Max
Raimbault	25	Arthur
Poincaré	45	Henri

Personne32(nom : chaîne, age : entier , prénom : chaîne) :
2ème forme normale

En pratique, il est aussi très difficile de faire vérifier automatiquement la mise en deuxième forme normale. Il faut trouver un jeu de données représentatif.

3ème forme normale :

Une relation est dite en troisième forme normale si chaque attribut qui ne compose pas la clef primaire, dépend directement de son identifiant et à travers une dépendance fonctionnelle. Les relations précédentes sont toutes en forme normale. Montrons un exemple de relation qui n'est pas en forme normale. Soit la relation **Personne4** (nom : chaîne, age : entier, civilité : Etatcivil, salaire : monétaire) où par exemple le salaire dépend de la clef primaire et que l'attribut civilité ne fait pas partie de la clef primaire (nom , âge) **:**

nom	âge	Civilité	salaire
Einstein	45	marié	1000
Lavoisier	41	marié	1000
Planck	52	veuf	1800
Raimbault	25	célibataire	1200
Poincaré	45	marié	1000

salaire = f (Civilité) => Pas 3ème forme normale

L'attribut salaire dépend de l'attribut civilité, ce que nous écrivons salaire = f(civilité), mais l'attribut civilité ne fait pas partie de la clef primaire clef = (nom , âge), donc **Personne4** n'est pas en 3ème forme normale **:**

Il faut alors décomposer la relation **Personne4** en deux relations **Personne41** et **Personne42** chacune en troisième forme normale:

Personne41 :

nom	âge	Civilité#
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Raimbault	25	célibataire
Poincaré	45	marié

Personne41 en 3ème forme normale, civilité clef étrangère

Personne42 :

Civilité	salaire
marié	1000
veuf	1800
célibataire	1200

Personne42 en 3ème forme normale, civilité clef primaire

En pratique, il est également très difficile de faire contrôler automatiquement la mise en troisième forme normale.

Remarques pratiques importantes pour le débutant :

- Les spécialistes connaissent deux autres formes normales. Dans ce cas le lecteur intéressé par l'approfondissement du sujet, trouvera dans la littérature, de solides références sur la question.

- Si la clef primaire d'une relation n'est composée que d'un seul attribut (choix conseillé lorsque cela est possible, d'ailleurs on trouve souvent des clefs primaires sous forme de numéro d'identification client, Insee,…) automatiquement, la relation est en 2ème forme normale, car chaque autre attribut non clef étrangère, ne dépend alors que de la valeur unique de la clef primaire.

- Penser dès qu'un attribut est fonctionnellement dépendant d'un autre attribut qui n'est pas la clef elle-même à décomposer la relation (créer deux nouvelles tables).

- En l'absence d'outil spécialisé, il faut de la pratique et être très systématique pour contrôler la normalisation.

Base de données relationnelles BD-R:

Ce sont des données structurées à travers :

- Une famille de domaines de valeurs,
- Une famille de relations n-aires,
- Les contraintes d'intégrité sont respectées par toute clef étrangère et par toute clef primaire.
- Les relations sont en 3ème forme normale. (à minima en 2ème forme normale)

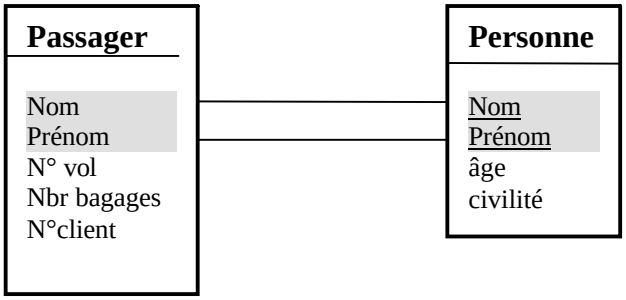
Les données sont accédées et manipulées grâce à un langage appelé **langage d'interrogation** ou **langage relationnel** ou **langage de requêtes**

Système de Gestion de Base de Données relationnel :

C'est une famille de logiciels comprenant :

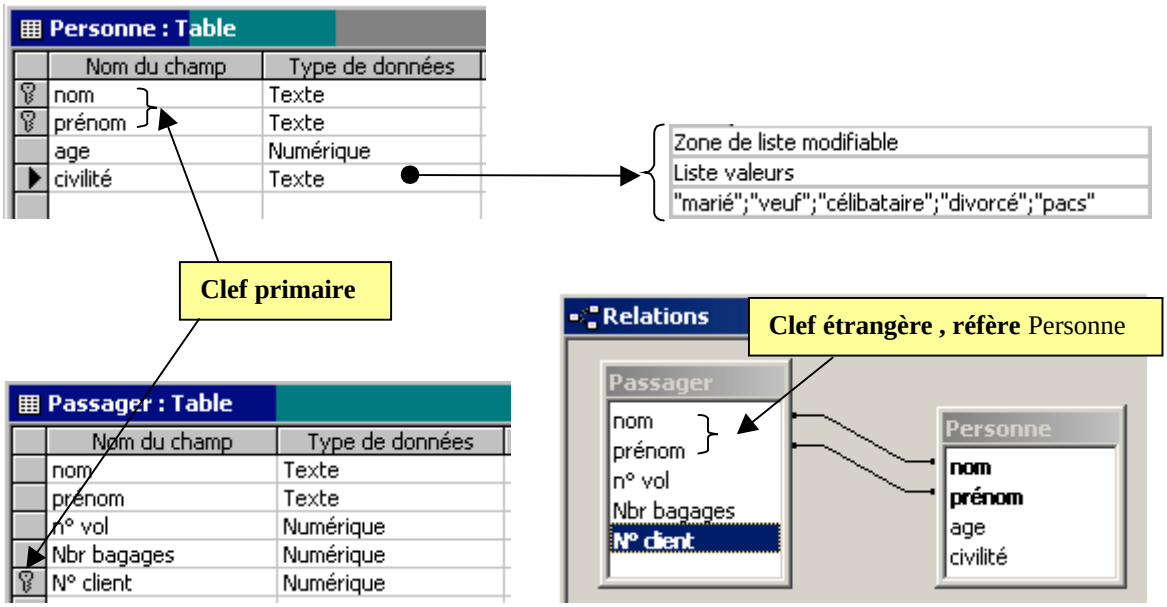
- Une BD-R.
- Un langage d'interrogation.
- Une gestion en interne des fichiers contenant les données et de l'ordonnancement de ces données.
- Une gestion de l'interface de communication avec les utilisateurs.
- La gestion de la sécurité des accès aux informations contenues dans la BD-R.

Le schéma relation d'une relation dans une BD relationnelle est noté graphiquement comme ci-dessous :

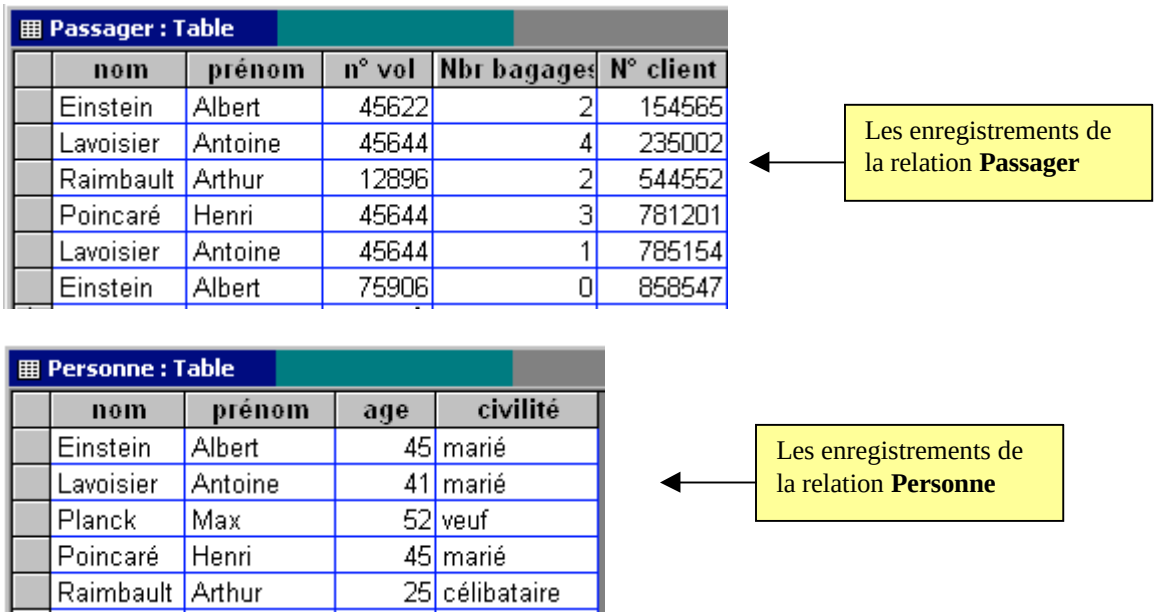


Les attributs reliés entre ces deux tables indiquent une liaison entre les deux relations.

Voici dans le **SGBD-R Access**, la représentation des schémas de relation ainsi que la liaison sans intégrité des deux relations précédentes **Passager** et **Personne** :



Access et la représentation des enregistrements de chaque table :



Passager : Table

nom	prénom	n° vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	4	235002
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Lavoisier	Antoine	45644	1	785154
Einstein	Albert	75906	0	858547

Personne : Table

nom	prénom	age	civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Poincaré	Henri	45	marié
Raimbault	Arthur	25	célibataire

Les besoins d'un utilisateur d'une base de données sont classiquement ceux que l'on trouve dans tout ensemble de données structurées : insertion, suppression, modification, recherche avec ou sans critère de sélection. Dans une BD-R, ces besoins sont exprimés à travers un langage d'interrogation. Historiquement deux classes de langages relationnels équivalentes en puissance d'utilisation et de fonctionnement ont été inventées : les langages **algébriques** et les langages des **prédicats**.

Un langage relationnel n'est pas un langage de programmation : il ne possède pas les structures de contrôle de base d'un langage de programmation (condition, itération, …). **Très souvent il doit être utilisé comme complément à l'intérieur de programmes Delphi, Java,...**.

Les langages d'interrogation prédicatifs sont des langages fondés sur la logique des prédicats du 1er ordre, le plus ancien s'appelle **Q**uery **B**y **E**xample QBE.

Ce sont les langages algébriques qui sont de loin les plus utilisés dans les SGBD-R du commerce, le plus connu et le plus utilisé dans le monde se dénomme le **S**tructured **Q**uery **L**anguage ou SQL. Un tel langage n'est qu'une implémentation en anglais d'opérations définies dans une algèbre relationnelle servant de modèle mathématique à tous les langages relationnels.

3. Principes fondamentaux d'une l'algèbre relationnelle

Une algèbre relationnelle est une famille d'opérateurs binaires ou unaires dont les opérandes sont des **relations**. Nous avons vu que l'on pouvait faire l'union, l'intersection , le produit cartésien de relations binaires dans un chapitre précédent, comme les relations n-aires sont des ensembles, il est possible de définir sur elle une algèbre opératoire utilisant les opérateurs classiques ensemblistes, à laquelle on ajoute quelques opérateurs spécifiques à la manipulation des données.

Remarque pratique :

> La phrase "**tous les n-uples sont distincts, puisqu'éléments d'un même ensemble nommé relation**" se transpose en pratique en la phrase **" toutes les lignes d'une même table nommée relation, sont distinctes** (même en l'absence de clef primaire explicite)".

Nous exhibons les opérateurs principaux d'une algèbre relationnelle et nous montrerons pour chaque opération, un exemple sous forme.

Union de 2 relations

> Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R ∪ Q de même degré et de même domaine contenant les enregistrements différents des deux relations R et Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

S = R ∪ Q

Q :

nom	âge	Civilité
Lupin	45	célibataire
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

Lupin	45	célibataire
Mozart	34	veuf
Gandhi	64	célibataire

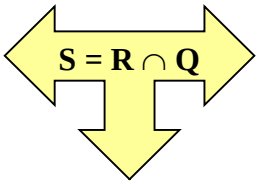
Remarque : (Planck, 52, veuf) ne figure qu'une seule fois dans la table R ∪ Q.

Intersection de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R ∩ Q de même degré et de même domaine contenant les enregistrements communs aux deux relations R et Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf



Q :

nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

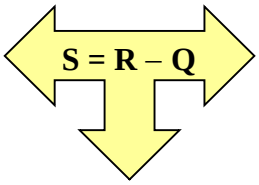
nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf

Différence de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R – Q de même degré et de même domaine contenant les enregistrements qui sont présents dans R mais qui ne sont pas dans Q (on exclut de R les enregistrements qui appartiennent à R ∩ Q) :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf



Q :

nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

nom	âge	Civilité
Lavoisier	41	marié

Produit cartésien de 2 relations

Soient R et Q deux relations de domaine et de degré quelconques (degré(R)=n, degré(Q)=p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

On peut calculer la nouvelle relation S = R x Q de degré n + p et de domaine égal à l'union des domaines de R et de Q contenant tous les couples d'enregistrements à partir d'enregistrements présents dans R et d'enregistrements présents dans Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

S = R x Q

Q :

ville	km
Paris	874
Rome	920

S :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Einstein	45	marié	Rome	920
Lavoisier	41	marié	Paris	874
Lavoisier	41	marié	Rome	920
Planck	52	Veuf	Paris	874
Planck	52	Veuf	Rome	920

Selection ou Restriction d'une relation

Soit R une relation, soit R($a_1 : E_1$, $a_2 : E_2$,…, $a_n : E_n$) le schéma de cette relation. Soit Cond(a_1, a_2 , … , a_n) une expression booléenne classique (expression construite sur les attributs avec les connecteurs de l'algèbre de Boole et les opérateurs de comparaison < , > , = , >= , <= , <>)

On note S = **select** (Cond(a_1, a_2 , … , a_n) , R), la nouvelle relation S construite ayant le même schéma que R soit S($a_1 : E_1$, $a_2 : E_2$,…, $a_n : E_n$), qui ne contient que les enregistrements de R qui satisfont à la condition booléenne Cond(a_1, a_2 , … , a_n).

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

Cond(a_1, a_2 , … , a_n) = { âge > 42 **et** ville=Paris }

S :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Gandhi	64	célibataire	Paris	258

Select ({ âge > 42 et ville=Paris }, R) signifie que l'on ne recopie dans S que les enregistrements de R constitués des personnes ayant séjourné à Paris et plus âgées que 42 ans.

Projection d'une relation

Soit R une relation, soit R($a_1 : E_1$, $a_2 : E_2$,…, $a_n : E_n$) le schéma de cette relation. On appelle S = **proj**(a_{k1} , a_{k2} … , a_{kp}) la projection de R sur un sous-ensemble restreint (a_{k1} , a_{k2} … , a_{kp}) avec p < n , de ses attributs, la relation S ayant pour

schéma le sous-ensemble des attributs $S(a_{k1} : E_{k1}, a_{k2} : E_{k2}, \ldots, a_{kp} : E_{kp})$ et contenant les enregistrements différents obtenus en ne considérant que les attributs $(a_{k1}, a_{k2} \ldots, a_{kp})$.

Exemple

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Lupin	42	Veuf	Paris	464
Einstein	45	marié	Venise	981
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

S1 = **proj**(nom , civilité)

nom	Civilité
Einstein	marié
Mozart	marié
Lupin	Veuf
Gandhi	célibataire
Lavoisier	marié
Planck	Veuf

S2 = **proj**(nom , âge)

nom	âge
Einstein	45
Mozart	32
Lupin	42
Gandhi	64
Lavoisier	41
Planck	52

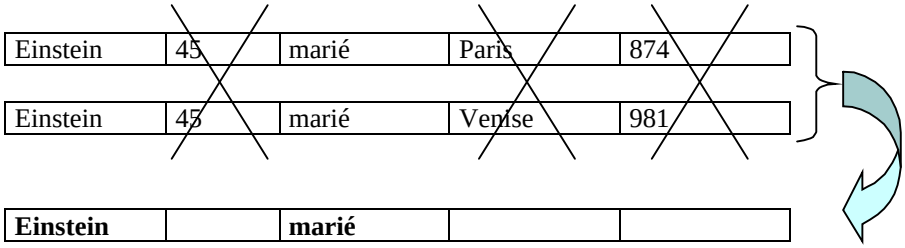
S3 = **proj**(nom , ville)

nom	ville
Einstein	Paris
Mozart	Rome
Lupin	Paris
Einstein	Venise
Gandhi	Paris
Lavoisier	Rome
Planck	Rome

S4 = **proj**(ville)

ville
Paris
Rome
Venise

Que s'est-il passé pour Mr Einstein dans S2 ?



Lors de la recopie des enregistrements de R dans S2 on a ignoré les attributs âge, ville et km, le couple (Einstein , marié) ne doit se retrouver qu'une seule fois car une relation est un ensemble et ses éléments sont tous distincts.

Jointure de deux relations

Soit Soient R et Q deux relations de domaine et de degré quelconques (degré(R) = n, degré(Q) = p), avec Domaine(R) $\cap$ Domaine(Q) = $\varnothing$ (pas d'attributs en communs).

soit R x Q leur produit cartésien de degré n + p et de domaine D union des domaines de R et de Q.
Soit un ensemble $(a_1, a_2, \ldots, a_{n+p})$ d'attributs du domaine D de R x Q.

> La relation **joint**(R,Q) = **select** (Cond(a_1, a_2 , … , a_{n+p}) , R x Q), est appelée jointure de R et de Q (c'est donc une sélection de certains attributs sur le produit cartésien).

Une jointure couramment utilisée en pratique, est celle qui consiste en la sélection selon une condition d'égalité entre deux attributs, les personnes de "l'art relationnel" la dénomment alors l'**équi-jointure**.

Exemples de 2 jointures :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

joint(R,Q)

Q :

ville	km
Paris	874
Rome	920

1°) S = **select** (km < 900 , R x Q)

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Lavoisier	41	marié	Paris	874
Planck	52	veuf	Paris	874

2°) S = **select** (km < 900 **et** Civilité = veuf, R x Q)

nom	âge	Civilité	ville	km
Planck	52	veuf	Paris	874

Nous nous plaçons maintenant du point de vue pratique, non pas de l'administrateur de BD mais de l'utilisateur uniquement concerné par l'extraction des informations contenues dans une BD-R.

Un SGBD permet de gérer une base de données. A ce titre, il offre de nombreuses fonctionnalités supplémentaires à la gestion d'accès simultanés à la base et à un simple interfaçage entre le modèle logique et le modèle physique : il sécurise les données (en cas de coupure de courant ou autre défaillance matérielle), il permet d'accéder aux données de manière confidentielle (en assurant que seuls certains utilisateurs ayant des mots de passe appropriés, peuvent accéder à certaines données), il ne permet de mémoriser des données que si elles sont du type abstrait demandé : on dit qu'il vérifie leur intégrité (des données alphabétiques ne doivent pas être enregistrées dans des emplacements pour des données numériques,...)

Actuellement, une base de données n'a pas de raison d'être sans son SGBD. Aussi, on ne manipule que des bases de données correspondant aux SGBD qui les gèrent : il vous appartient de choisir le SGBD-R qui vous convient (il faut l'acheter auprès de vendeurs qui généralement, vous le fournissent avec une application de manipulation visuelle, ou bien utiliser les SGBD-R qui vous sont livrés gratuitement avec certains environnements de développement comme Delphi ou Visual Studio ou encore utiliser les produits gratuits comme mySql).

Lorsque l'on parle d'utilisateur, nous entendons l'application utilisateur, car l'utilisateur final n'a pas besoin de connaître quoique ce soit à l'algèbre relationnelle, il suffit que l'application utilisateur communique avec lui et interagisse avec le SGBD.

Une application doit pouvoir "parler" au SGBD : elle le fait par le moyen d'un langage de manipulation des données. Nous avons déjà précisé que la majorité des SGBD-R utilisait un langage relationnel ou de requêtes nommé SQL pour manipuler les données.

4. SQL et Algèbre relationnelle

Requête

Les requêtes sont des questions posées au SGBD, concernant une recherche de données contenues dans une ou plusieurs tables de la base.

Par exemple, on peut disposer d'une table définissant des clients (noms, prénoms, adresses, n°de client) et d'une autre table associant des numéros de clients avec des numéros de commande d'articles, et vouloir poser la question : quels sont les noms des clients ayant passé des commandes ?

Une requête est en fait, une instruction de type langage de programmation, respectant la norme SQL, permettant de réaliser un tel questionnement. L'exécution d'une requête permet d'extraire des données en provenance de tables de la base de données : ces données réalisent ce que l'on appelle une **projection** de champs (en provenance de plusieurs tables). Le résultat d'exécution d'une requête est une table constituée par les réponses à la requête.

Le SQL permet à l'aide d'instructions spécifiques de manipuler des données à l'intérieur des tables :

Instruction SQL	Actions dans la (les) table(s)
INSERT INTO <…>	Ajout de lignes
DELETE FROM <…>	Suppression de lignes
TRUNCATE TABLE <…>	Suppression de lignes
UPDATE <…>	Modification de lignes
SELECT <…>**FROM** <…>	Extraction de données

Ajout, suppression et modification sont les trois opérations typiques de la **mise à jour** d'une BD. L'extraction concerne la **consultation** de la BD.

Il existe de nombreuses autres instructions de création, de modification, de suppression de tables, de création de clefs, de contraintes d'intégrités référentielles, création d'index, etc… Nous nous attacherons à donner la traduction en SQL des opérateurs principaux de l'algèbre relationnelle que nous venons de citer.

Traduction en SQL des opérateurs relationnels

C'est l'instruction SQL "**SELECT** <…>**FROM** <…>" qui implante tous ces opérateurs. Tous les exemples utiliseront la relation R = TableComplete suivante et l'interpréteur SQL d'**Access** :

TableComplete : Table

nom	âge	civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Lupin	42	Veuf	Paris	464
Einstein	45	marié	Venise	981
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

La relation initiale : R=TableComplete

Projection d'une relation R

S = **proj**(a_{k1} , a_{k2} … , a_{kp})

SQL : **SELECT DISTINCT** a_{k1} , a_{k2} … , a_{kp} **FROM** R

Instruction SQL version opérateur algèbre :
SELECT DISTINCT nom , civilité **FROM** Tablecomplete

Lancement de la requête SQL :

RequêteS1 : Requête Sélection

```
SELECT DISTINCT [nom], [civilité]
FROM TableComplete;
```

Le mot DISTINCT assure que l'on obtient bien une nouvelle relation.

Table obtenue après requête :

RequêteS1 : Requête Sélection

nom	civilité
Einstein	marié
Gandhi	célibataire
Lavoisier	marié
Lupin	Veuf
Mozart	marié
Planck	Veuf

Instruction SQL non relationnelle (tout même redondant) :
SELECT nom , civilité **FROM** Tablecomplete

Lancement de la requête SQL :

RequêteS1 : Requête Sélection

```
SELECT [nom], [civilité]
FROM TableComplete;
```

Table obtenue après requête :

RequêteS1 : Requête Sélé

nom	civilité
Einstein	marié
Mozart	marié
Lupin	Veuf
Einstein	marié
Gandhi	célibataire
Lavoisier	marié
Lupin	Veuf
Planck	Veuf

Remarquons dans le dernier cas **SELECT** nom , civilité **FROM** Tablecomplete que la table obtenue n'est qu'une extraction de données, mais qu'en aucun cas elle ne constitue une relation puisqu'une relation est un ensemble et que les enregistrements sont tous distincts!

Une autre projection sur la même table :

Instruction SQL version opérateur algèbre :
SELECT DISTINCT nom , ville **FROM** Tablecomplete

Lancement de la requête SQL :

Requête52 : Requête Sélection

```
SELECT DISTINCT [nom], [ville]
FROM TableComplete;
```

Table obtenue après requête :

Requête52 : Requête Sél

	nom	ville
	Einstein	Paris
	Einstein	Venise
	Gandhi	Paris
	Lavoisier	Rome
	Lupin	Paris
	Mozart	Rome
▶	Planck	Rome

Sélection-Restriction

S = **select** (Cond(a_1, a_2 , … , a_n) , R)

SQL : **SELECT * FROM** R **WHERE** Cond(a_1, a_2 , … , a_n)

*Le symbole * signifie toutes les colonnes de la table (tous les attributs du schéma)*

Instruction SQL version opérateur algèbre :
SELECT * FROM Tablecomplete **WHERE** âge > 42 **AND** ville = Paris

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection

```
SELECT * FROM TableComplete
WHERE [âge]>42 AND [ville]="Paris";
```

Table obtenue après requête :

TableComplete Requête : Requête Sélection

	nom	âge	civilité	ville	km
	Einstein	45	marié	Paris	874
	Gandhi	64	célibataire	Paris	258

On a sélectionné toutes les personnes de plus de 42 ans ayant séjourné à Paris.

Combinaison d'opérateur projection-restriction

Projection distincte et sélection :
SELECT DISTINCT nom , civilité, âge **FROM** Tablecomplete **WHERE** âge >= 45

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection

```
SELECT DISTINCT [nom], [civilité], [âge]
FROM TableComplete
WHERE [âge]>=45 ;
```

Table obtenue après requête :

TableComplete Requête : Requête Sé

	nom	civilité	âge
	Einstein	marié	45
	Gandhi	célibataire	64
▶	Planck	Veuf	52

On a sélectionné toutes les personnes d'au moins 45 ans et l'on ne conserve que leur nom, leur civilité et leur âge.

Produit cartésien

S = **R x Q**

SQL : **SELECT * FROM** R , Q

Afin de ne pas présenter un exemple de table produit trop volumineuse, nous prendrons comme opérandes du produit cartésien, deux tables contenant peu d'enregistrements :

Personne : Table

nom	prénom	age	civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié

Employeur : Table

nom	n°Insee
Université	12112472
Collège	25478550
Institut	54578559

Produit cartésien :
SELECT * FROM Employeur , Personne

Employeur Requête : Requête Sélection

Personne.nom	prénom	age	civilité	Employeur.nom	n°Insee
Einstein	Albert	45	marié	Université	12112472
Lavoisier	Antoine	41	marié	Université	12112472
Einstein	Albert	45	marié	Collège	25478550
Lavoisier	Antoine	41	marié	Collège	25478550
Einstein	Albert	45	marié	Institut	54578559
Lavoisier	Antoine	41	marié	Institut	54578559

Nous remarquons qu'en apparence l'attribut **nom** se retrouve dans le domaine des deux relations ce qui semble contradictoire avec l'hypothèse "Domaine(R) $\cap$ Domaine(Q) = $\varnothing$ (pas d'attributs en communs). En fait ce n'est pas un attribut commun puisque les valeurs sont différentes, il s'agit plutôt de deux attributs différents qui ont la même identification. Il suffit de préfixer l'identificateur par le nom de la relation (Personne.nom et Employeur.nom).

Combinaison d'opérateurs : projection - produit cartésien

SELECT Personne.nom , prénom, civilité, n°Insee **FROM** Employeur , Personne

On extrait de la table produit cartésien uniquement 4 colonnes :

Employeur Requête : Requête Sélection

nom	prénom	civilité	n°Insee
Einstein	Albert	marié	12112472
Lavoisier	Antoine	marié	12112472
Einstein	Albert	marié	25478550
Lavoisier	Antoine	marié	25478550
Einstein	Albert	marié	54578559
Lavoisier	Antoine	marié	54578559

Intersection, union, différence,

S = **R** $\cap$ **Q**
SQL : **SELECT * FROM** R **INTERSECT SELECT * FROM** Q

S = **R** $\cup$ **Q**
SQL : **SELECT * FROM** R **UNION SELECT * FROM** Q

S = **R** – **Q**
SQL : **SELECT * FROM** R **MINUS SELECT * FROM** Q

Jointure de deux relations

Soient R et Q deux relations de domaine et de degré quelconques (degré(R) = n, degré(Q) = p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

La jointure **joint**(R,Q) = **select** (Cond(a_1, a_2 , … , a_{n+p}) , R x Q).

SQL : **SELECT * FROM** R ,Q **WHERE** Cond(a_1, a_2 , … , a_{n+p})

Employeur : Table

nom	n°Insee
Université	12112472
Collège	25478550
Institut	54578559

Passager : Table

nom	prénom	n° vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	4	235002
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Lavoisier	Antoine	45644	1	785154
Einstein	Albert	75906	0	858547
		0	0	0

Employeur RequêteJoin : Requête Sélection

```
SELECT *
FROM Employeur, Passager
WHERE Passager.[n° vol] > 20000;
```

Employeur RequêteJoin : Requête Sélection

Employeur.no	n°Insee	Passager.	prénom	n° vol	Nbr bagages	N° client
Université	12112472	Einstein	Albert	45622	2	154565
Université	12112472	Lavoisier	Antoine	45644	1	785154
Université	12112472	Poincaré	Henri	45644	3	781201
Université	12112472	Einstein	Albert	75906	0	858547
Université	12112472	Lavoisier	Antoine	45644	4	235002
Collège	25478550	Einstein	Albert	45622	2	154565
Collège	25478550	Lavoisier	Antoine	45644	1	785154
Collège	25478550	Poincaré	Henri	45644	3	781201
Collège	25478550	Einstein	Albert	75906	0	858547
Collège	25478550	Lavoisier	Antoine	45644	4	235002
Institut	54578559	Einstein	Albert	45622	2	154565
Institut	54578559	Lavoisier	Antoine	45644	1	785154
Institut	54578559	Poincaré	Henri	45644	3	781201
Institut	54578559	Einstein	Albert	75906	0	858547
Institut	54578559	Lavoisier	Antoine	45644	4	235002

Remarque pratique importante

Le langage SQL est plus riche en fonctionnalités que l'algèbre relationnelle. En effet SQL intègre des possibilités de calcul (numériques et de dates en particulier).

Soit une table de tarifs de produit avec des prix hors taxe:

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

1°) Usage d'un opérateur multiplicatif : calcul de la nouvelle table des tarifs TTC abondés de la TVA à 20% sur le prix hors taxe.

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Tarifs TTc

Tarifs TTc : Requête Sélection

```
SELECT Article , PrixHT*1.20 AS PrixTTC
FROM Tarifs;
```

Tarifs TTc : Requête Sélection

Article	PrixTTC
chaise	12
table	120
Téléviseur	1200

2°) Usage de la fonction intégrée SUM : calcul du total des prix HT.

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Total HT

Total HT : Requête Sélection

```
SELECT SUM(PrixHT) AS Total
FROM Tarifs;
```

Total HT : Requête Sélection

Total
1 110,00 €

5. Exemple de communication entre Delphi et les BD

Chaque langage permet d'une façon plus ou moins simple, d'accéder à une BD. Ce qui revient à dire que chaque constructeur de langage met en place sa propre solution pour qu'une application s'interface avec un SGBD à travers des commandes SQL (on appelle cela une solution propriétaire). Du côté de l'utilisateur, l'application rend le plus transparent possible la navigation sous SQL.

Delphi de Borland propose et a proposé des solutions propriétaires qui évoluent au cours du temps et des versions. En outre, les SGBD évoluent eux aussi, ce qui rend quasi impossible un choix simple et portable d'une solution standard. Enfin, il y a une grande différence de fonctionnalités entre un SGBD comme Access et Oracle ce qui ne va pas dans le sens de l'interopérabilité.

Principe général à adopter lorsque l'on veut écrire une application qui accède à une base déjà existante. Voir dans la documentation du langage si la version du SGBD est supportée par la version du langage Delphi que vous comptez utiliser. Ensuite, la documentation propose une stratégie de communication (utilisation de certaines classes) plus adaptée au SGBD.

Il existe parmi les choix proposés par Delphi, un moyen d'accès qui se nomme le BDE (Borland Data Engine) qui, bien qu'il soit en fin de vie est présent dans les versions 5, 6 et 7 professionnelle ou architecte, entreprise client-serveur de Delphi. Les versions perso ne contiennent rien qui permet d'accéder aux bases de données. O.Dahan et P.Thot dans leur excellent ouvrage "Applications professionnelles Delphi 7 studio" parlent du BDE en tant qu'outil pour le monde professionnel : *"..le BDE n'est plus une solution d'avenir. Toutefois enrichi par des années d'évolution, ce produit sait rendre des services qui ne peuvent être satisfaits par les autres solutions disponibles. Connaître ces avantages peut vous sauver la mise dans certaines situations.*"

Nous supposerons que vous disposez d'Access97 (contenu dans Office pro97), au minimum une base Access déjà existante et une version de Delphi contenant le BDE.

L'organisation physique d'une base de données dépend du SGBD qui la gère : certaines bases sont physiquement représentées par un seul fichier dans lequel sont stockées toutes les tables, requêtes,... (Microsoft Access fait cela par exemple), d'autres sont physiquement représentées par un dossier sur disque, et les tables, requêtes,... sont stockées sous la forme de fichiers séparés dans ce dossier.

5.1 Principe du BDE

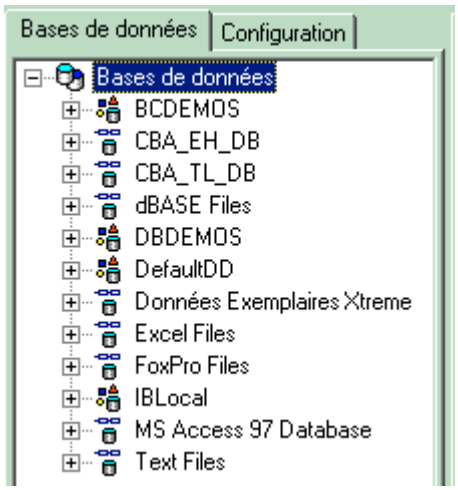
Vous pouvez accéder au BDE par le Panneau de Configuration en lançant "Administrateur BDE". Le BDE permet la communication de Delphi avec les Bases de données. Borland utilise un ensemble de DLL dans lesquelles sont codées les SGBD.

Pour pouvoir gérer des SGBD d'autres types (par exemple Microsoft Access) vous devez avoir acheté les DLL nécessaires (par exemple si vous avez acheté Microsoft Access, la DLL

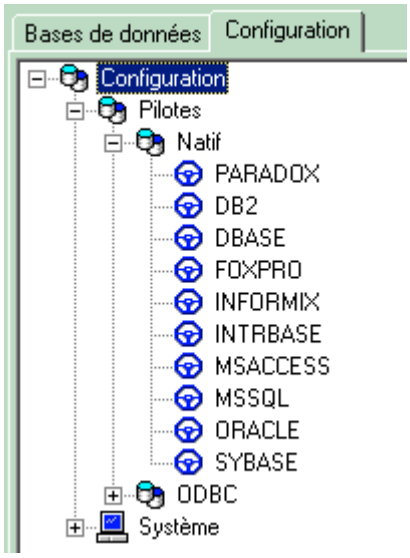
nécessaire est IDDA3532.DLL, qui est automatiquement installée sous Windows lorsque vous installez Access, et le BDE pourra l'utiliser).

Il y a 2 sections dans le BDE :

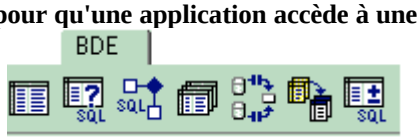
- Base de données : dans cette section, on déclare des alias de bases de données (c'est à dire des noms fictifs pour une base de données physique), et on indique sa position physique sur disque, et le type de SGBD qui peut la gérer (le type de pilote dans notre cas, qui est associé à un SGBD).



- Configuration : c'est là que les noms des pilotes sont déclarés, et associés physiquement aux DLL qui permettent de gérer le SGBD associé à un type donné.

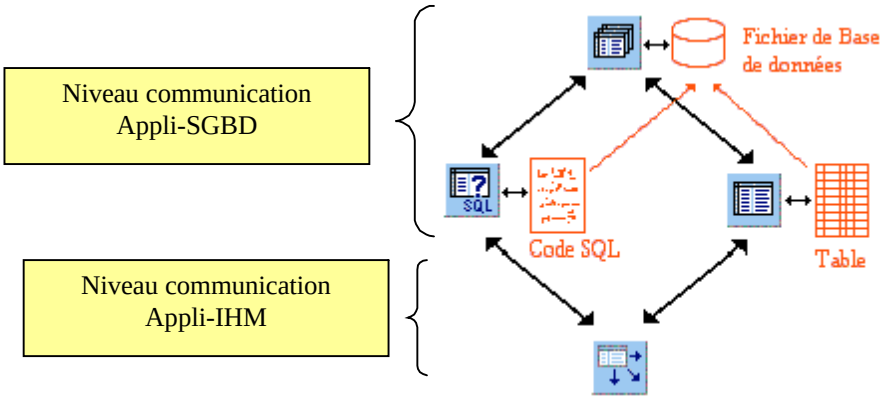


5.2. Les classes Delphi de communication avec les bases de données

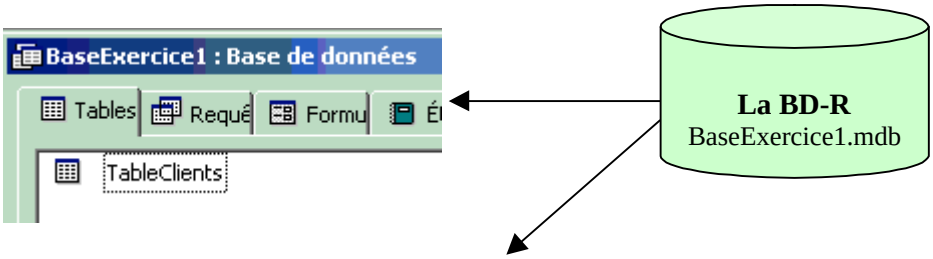
Trois classes fondamentales pour qu'une application accède à une BD *onglet BDE de la palette :* 	
TQuery	*Permet d'effectuer des requêtes dans une BD*
TDataBase	*Permet de connecter physiquement une application Delphi à un fichier de BD*
TTable	*Permet d'accéder à une table de la BD*

Une classe pour communiquer avec les classes précédentes et l'utilisateur *onglet ControlesDB de la palette :*	
TDataSource	*Permet aux composants de navigations (ceux de l'onglet "ControlesDB" de la palette de Delphi, d'accéder aux objets fournis par un Ttable ou un TQuery*

Schéma de communication dans une application accédant à une BD :



Exemple : soit une BD Access nommée BaseExercice1.mdb, contenant une table nommée TableClients avec 4 enregistrements :

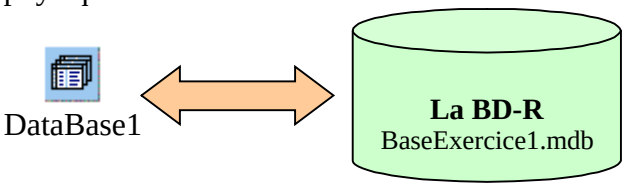


La TableClients est composée de 4 enregistrements :

TableClients : Table

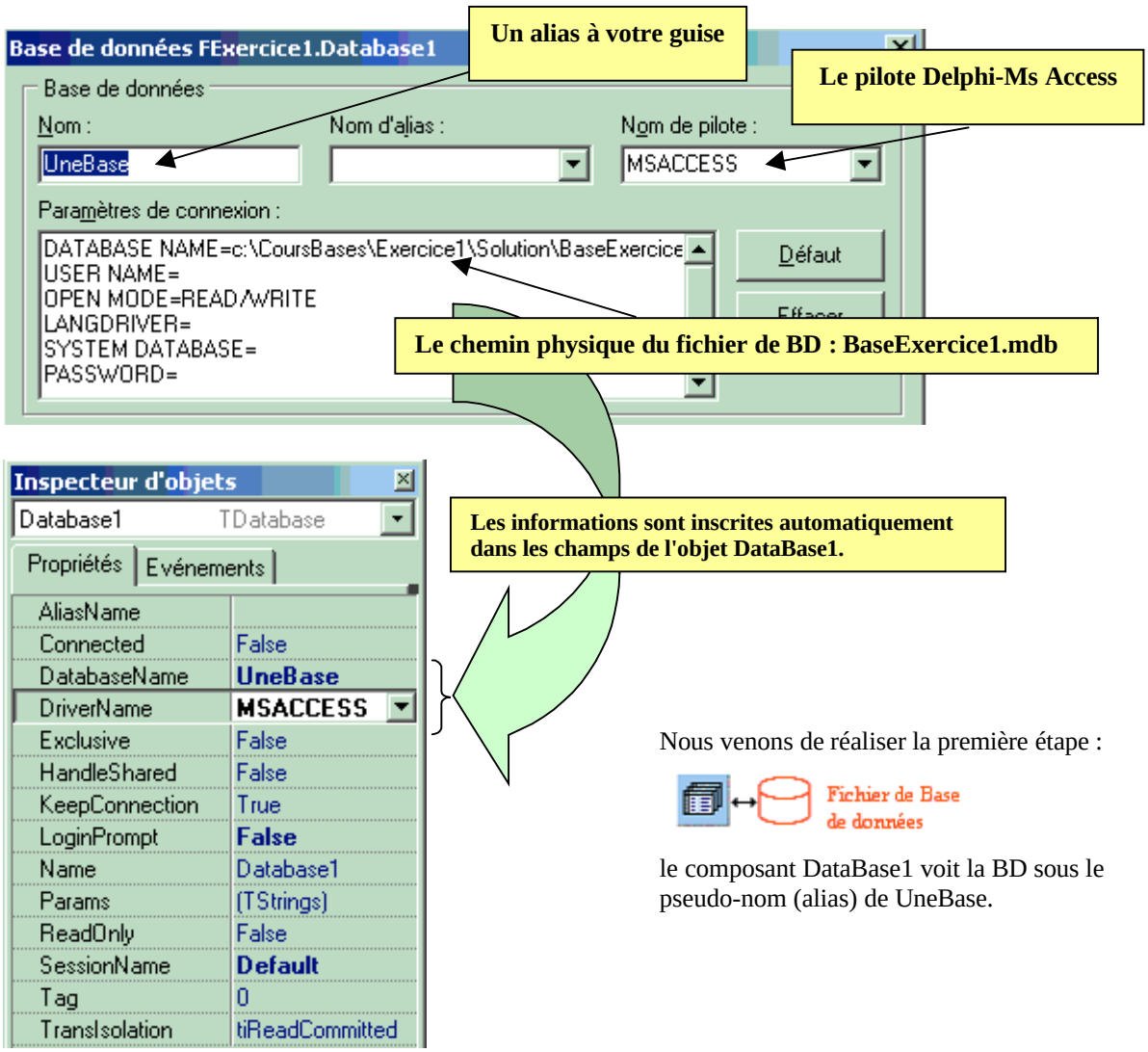
Nom	Prénom	Adresse	CP	Ville
ALKARTI	Joseph	3 route des Merles	54023	Cartes
BOULAIS	Sylvie	5 allée des Pins	05123	Teluies
GANZ	Karl	27 avenue St Laurent	78129	Isyfes
PROUDON	Amélie	105 rue des Platanes	45123	Mentisse

Dans l'application Delphi, il faut déposer un composant TDataBase que nous relions à la BD physique :



On double clique sur le composant DataBase1et on obtient une fenêtre de dialogue permettant la connexion physique.

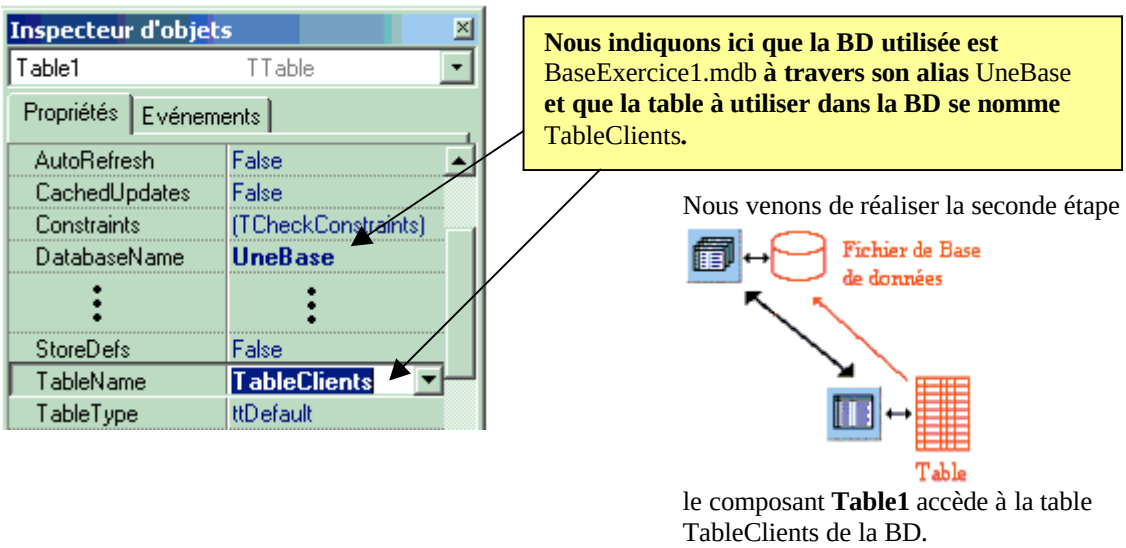
La fenêtre de dialogue obtenue :



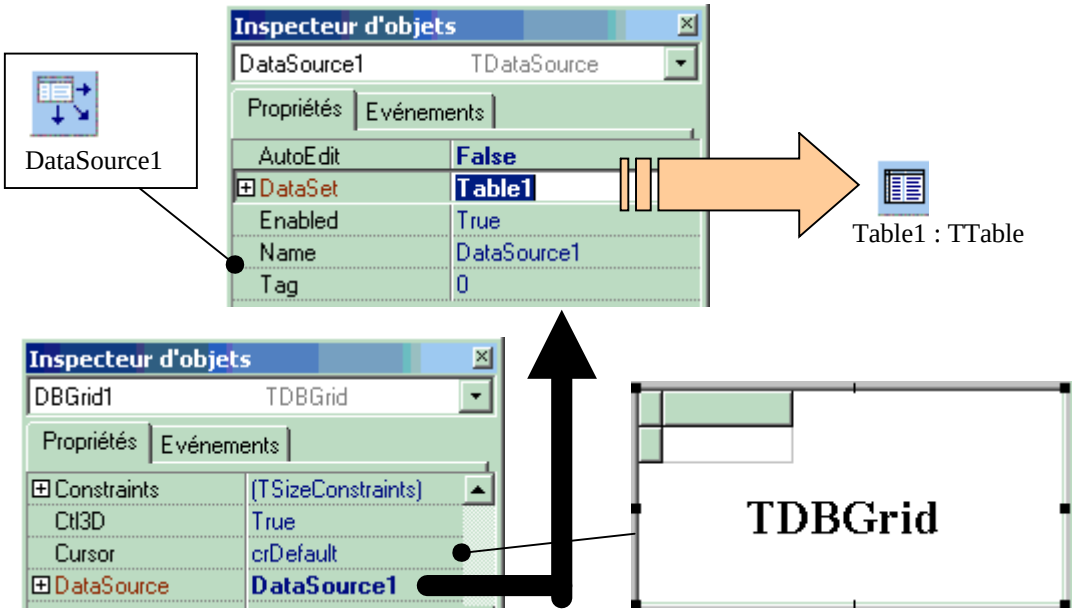
Nous venons de réaliser la première étape :

le composant DataBase1 voit la BD sous le pseudo-nom (alias) de UneBase.

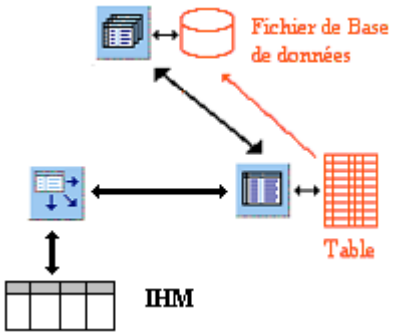
Dans la seconde étape du travail, nous devons donner la possibilité à notre application de travailler avec une table de la BD, en l'occurrence ici, avec la table TableClients de **BaseExercice1.mdb**, nous déposons un composant **Table1** de type TTable pour cette opération :



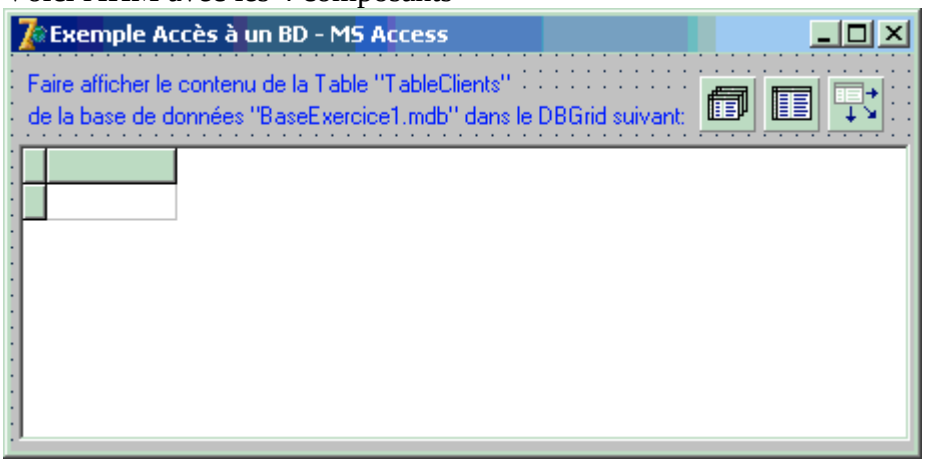
Nous terminons par la partie IHM de l'application : nous utilisons un composant de navigation TDBGrid qui affiche et manipule les enregistrements d'une table dans une grille tabulaire. Il n'est pas directement connecté au composant Table1, nous avons vu que c'est la classe TDataSource qui fait la liaison entre le composant d'IHM (ici nous avons choisi TDBGrid) et le TTable connecté à la table TableClients dans la BD :



Nous avons fini le processus de mise en place de la connexion de l'application et de la navigation dans la table TableClients de la BD :

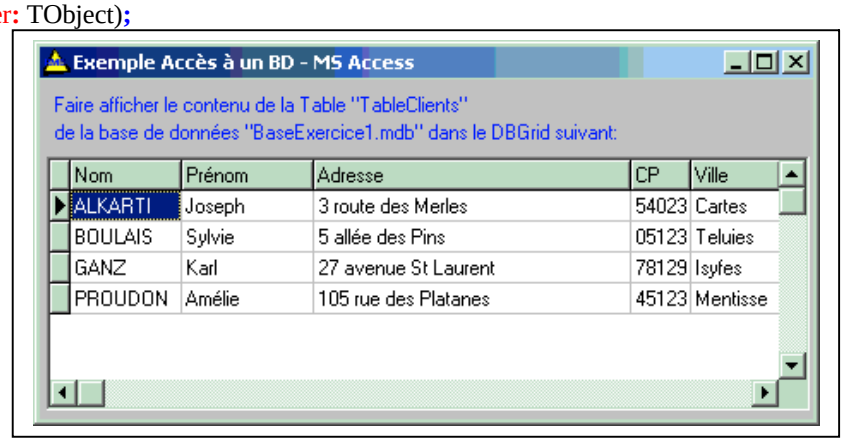


Voici l'IHM avec les 4 composants



Code source Delphi 5 , 6, 7

```
unit uFExercice1;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, Grids, DBGrids, StdCtrls, Mask, DBTables, Db;

 type
  TFExercice1 = class(TForm)
  DBGrid1: TDBGrid;
  Table1: TTable;
  DataSource1: TDataSource;
  Database1: TDatabase;
  Label1: TLabel;
  Label2: TLabel;
  procedure FormCreate(Sender: TObject);
 private
 { Déclarations privées }
 public
 { Déclarations publiques }
end;

 var
  FExercice1: TFExercice1;
  App_Path:string;

implementation

 {$R *.DFM}

 procedure TFExercice1.FormCreate(Sender: TObject);
 begin
  App_Path:=extractfilepath(application.exename);
  DataBase1.params[0]:='DATABASE NAME='+App_Path+'BaseExercice1.mdb';
  //paramètre dynamiquement le chemin de la base
  //de données (les lignes vides de paramètre doivent
  //déjà exister)
  //ou alternativement à la ligne précédente:
  {DataBase1.params.values['DATABASE NAME']:=App_Path+'BaseExercice1.mdb';}
```



```
 DataBase1.connected:=true;  //connexion sur la base de données
 //(évidemment il faut d'abord paramétrer
 //correctement le composant DataBase1)
 Table1.open; //Ouverture de la table attachée au composant Table1
              //(il faut avoir bien paramétré le composant Table1auparavant
end;

end.
```

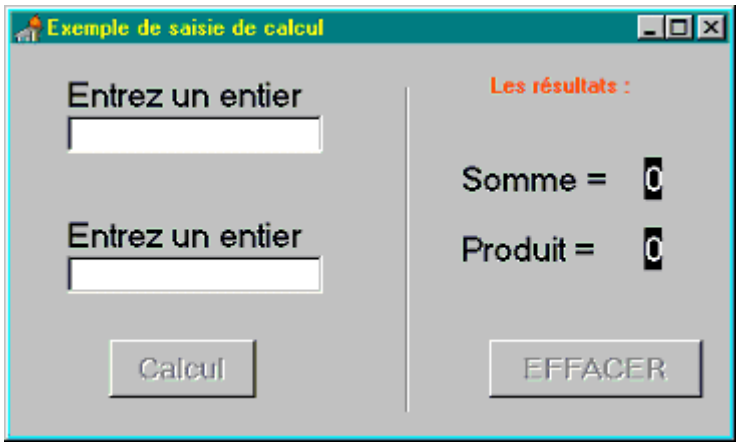
Exercices chapitre 7

Ex-1 : pilotage récapitulatif

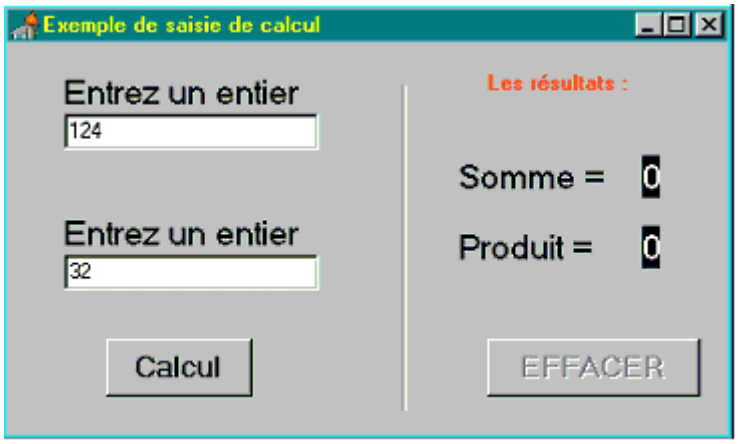
Nous voulons construire une interface permettant la saisie par l'utilisateur de deux entiers et l'autorisant à effectuer leur somme et leur produit uniquement lorsque les entiers sont entrés tous les deux. Aucune sécurité n'est apportée **pour l'instant** sur les données.

Voici l'état visuel de l'interface au lancement du programme :

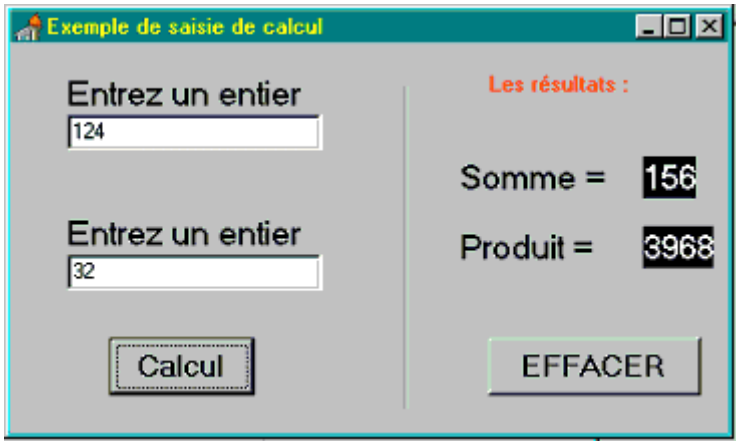
2 zones de saisie
2 boutons d'actions
2 zones d'affichages des résultats



Dès que les deux entiers sont entrés, le bouton **Calcul** est activé:



Lorsque l'on clique sur le bouton **Calcul**, le bouton **EFFACER** est activé et les résultats s'affichent dans leurs zones respectives :



Un clic sur le bouton **EFFACER** ramène l'interface à l'état initial.

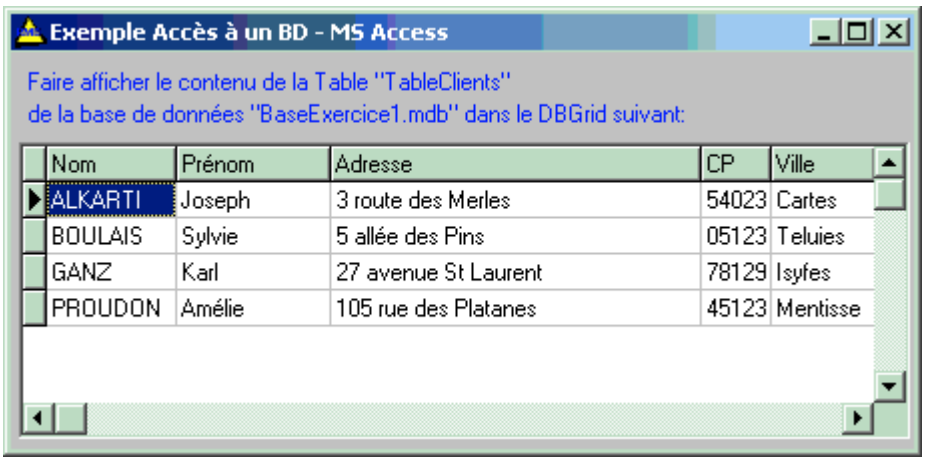
Exercices BD et Delphi

On donne la BD nommée BaseExercice.mdb et la relation TableClients ci-dessous :

TableClients : Table

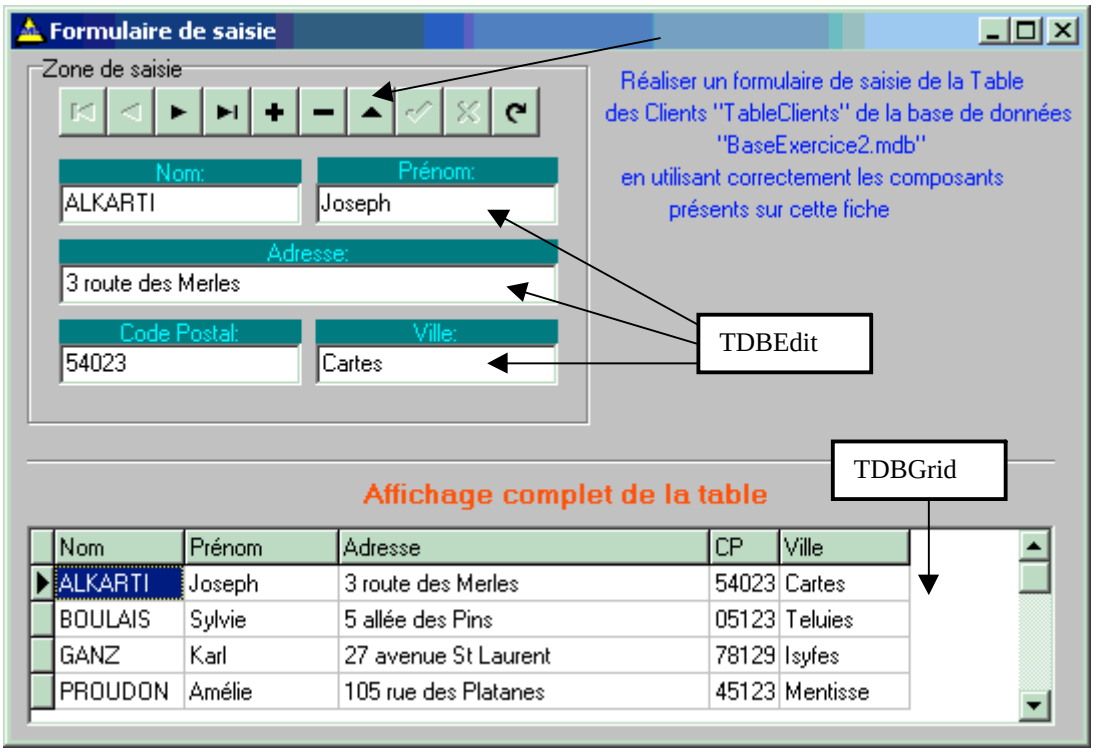
Nom	Prénom	Adresse	CP	Ville
ALKARTI	Joseph	3 route des Merles	54023	Cartes
BOULAIS	Sylvie	5 allée des Pins	05123	Teluies
GANZ	Karl	27 avenue St Laurent	78129	Isyfes
PROUDON	Amélie	105 rue des Platanes	45123	Mentisse

Ex-2 : construire une IHM Delphi permettant d'afficher la table TableClients dans un TDBGrid , comme ci-desssous :



Ex-3 : construire une IHM Delphi de formulaire de saisie la table TableClients en utilisant les composant TDBEdit, TDBNavigator, TDBGrid , comme ci-desssous ::

TDBNavigator

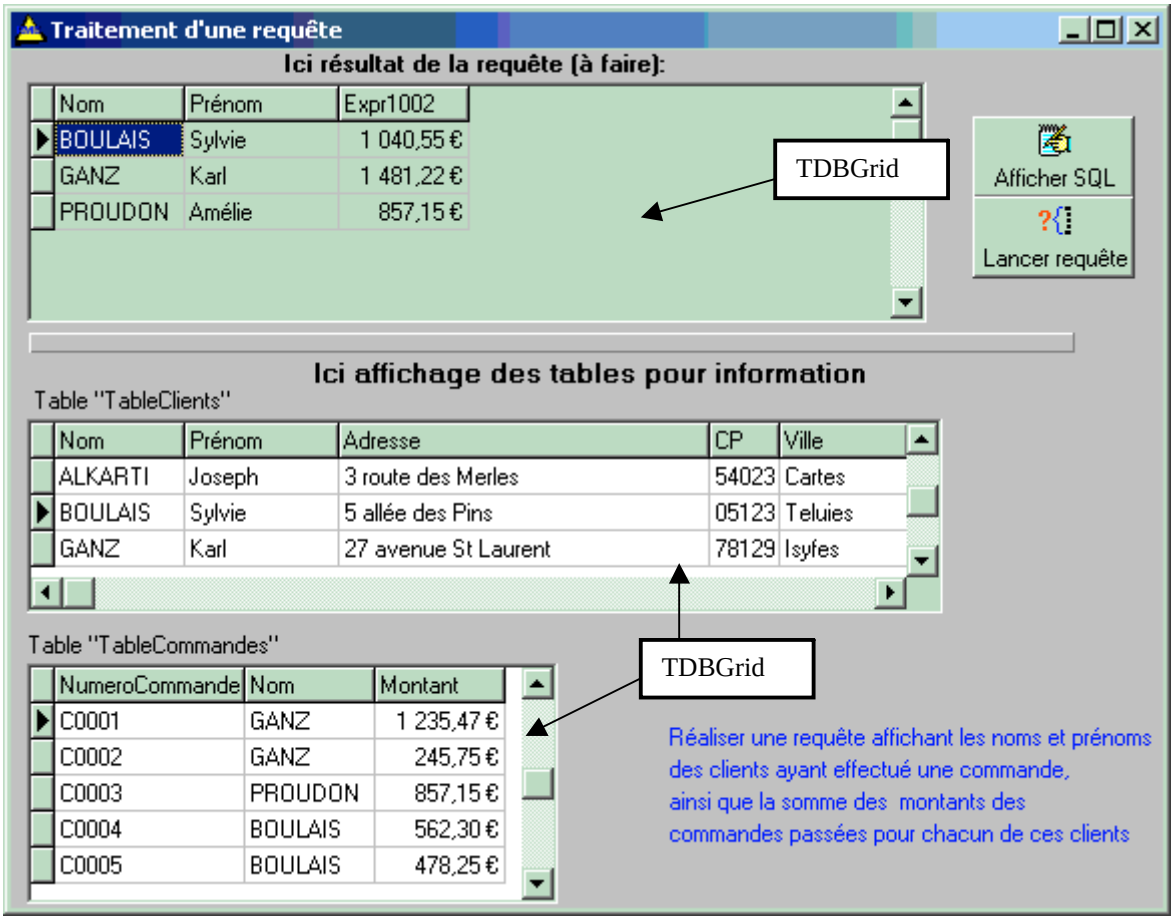


Ex-4 : construire une IHM Delphi de lancement d'une requête sur la table TableClients et une autre TableCommandes en utilisant le composant Tquery.

La commande SQL à lancer est la suivante :

```
SELECT TableClients.Nom, TableClients.Prénom, Sum(TableCommandes.Montant)
FROM TableClients INNER JOIN TableCommandes ON TableClients.Nom = TableCommandes.Nom
GROUP BY TableClients.Nom, TableClients.Prénom
```

IHM à programmer :



Ex-5 : Soit une BD contenant des informations de produits vendus dans un commerce. La BD contient deux tables, une table magasin :

Magasin : Table

CodeArticle	DésignArticle	QuantitéStock	SeuilAlerte
A0001	Bottes	157	20
A0002	Brosse poil dur	56	60
A0003	Savon doux	78	50
A0004	Trousse secours	6	10
A0005	Cirage noir	46	15
A0006	Ampoule 40W	13	20

Et une table PrixArticles:

PrixArticles : Table

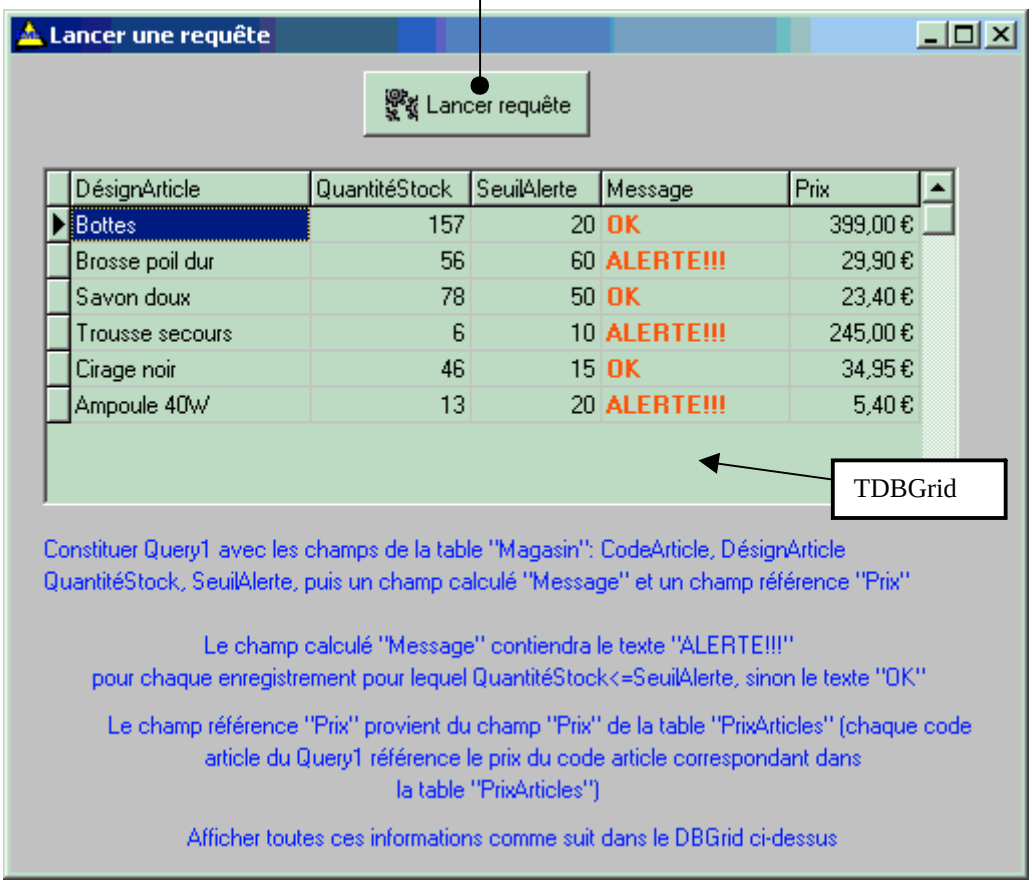
CodeArticle	Prix
A0001	399,00 F
A0002	29,90 F
A0003	23,40 F
A0004	245,00 F
A0005	34,95 F
A0006	5,40 F

Le champ CodeArticle est une clef primaire de chaque table

La commande SQL à lancer est la suivante :

```
SELECT Magasin.CodeArticle, Magasin.DésignArticle, Magasin.QuantitéStock, Magasin.SeuilAlerte
FROM Magasin  ORDER BY Magasin.CodeArticle
```

Construire une IHM Delphi de lancement de la requête ci-dessus sur la table Magasin en utilisant le composant TQuery.



Ex-6 : Expressions arithmétiques et arbre binaire : nous détaillons dans cet exemple, la démarche de création d'un programme ayant pour but de construire et de parcourir un arbre de syntaxe abstrait des expressions arithmétiques fondées sur une C-grammaire.

Nous procédons par étapes séparées afin de bien faire comprendre le processus de conception du programme. La première étape consiste à écrire un programme d'analyse des expressions permettant de repérer les variables représentées par des lettres et les opérateurs. Il ne s'agit pas ici d'un processus lexical, mais d'un processus syntaxique déjà abordé dans le cours.

Une fois le programme en version analyse syntaxique élaboré, nous passons à l'étape de génération de l'arbre binaire abstrait représentant l'expression en cours d'analyse.

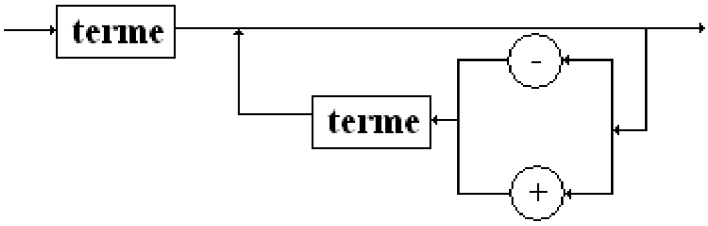
Grammaire des expressions utilisée :

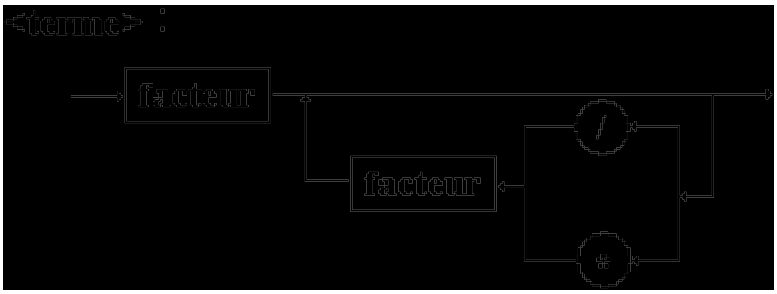
Vn = {<expr>, <terme>, <facteur>, <caractère> }
Vt = { **+** ,**-** , **(** , **)** ,***** , **/** , **a** ,..., **z**}

Axiome : <expr>

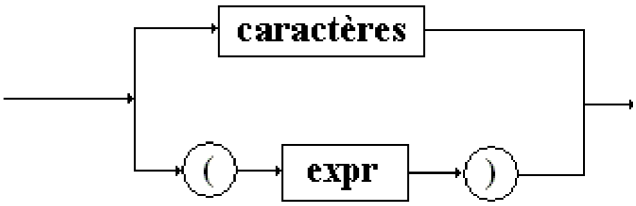
Règles :

<expr> :





<facteur> :



<caractères> ::= a| b | c | ..... | z

Solution des exercices

Ex-1 : pilotage récapitulatif- solution complète détaillée.

Graphe événementiel complet de l'interface

Avec comme conventions sur les messages :

X_1 = exécuter les calculs sur les valeurs entrées.

X_2 = exécuter l'effacement des résultats.

M_1 = message à l'edit1 de tout effacer.

M_2 = message à l'edit2 de tout effacer.

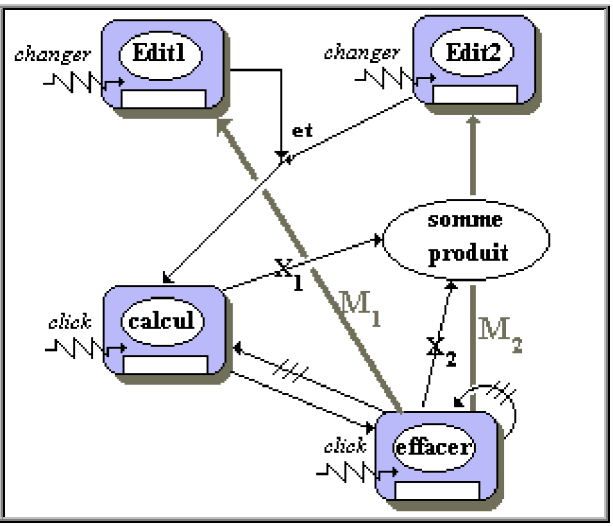


Table des actions événementielles associées au graphe

Changer Edit1	Calcul activable (si changer Edit2 a eu lieu)
Changer EDIT2	Calcul activable (si changer Edit1 a eu lieu)
Clic CALCUL	Exécuter X1 EFFACER activable Afficher les résultats
Clic EFFACER	EFFACER désactivé CALCUL désactivé Message M1 Message M2

Table des états initiaux des objets sensibles aux événements

Edit1	***activé***
Edit2	***activé***
Buttoncalcul	***désactivé***
Buttoneffacer	***désactivé***

- Nous voulons disposer d'un bouton permettant de lancer le calcul lorsque c'est possible et d'un bouton permettant de tout effacer. Les deux boutons seront désactivés au départ.
- Nous choisissons 6 objets visuels : deux TEdit, **Edit1** et **Edit2** pour la saisie ; deux Tbutton, **ButtonCalcul** et **Buttoneffacer** pour les changements de plans d'action ; deux Tlabel **LabelSomme** et **LabelProduit** pour les résultats

Les objets visuels Delphi choisis

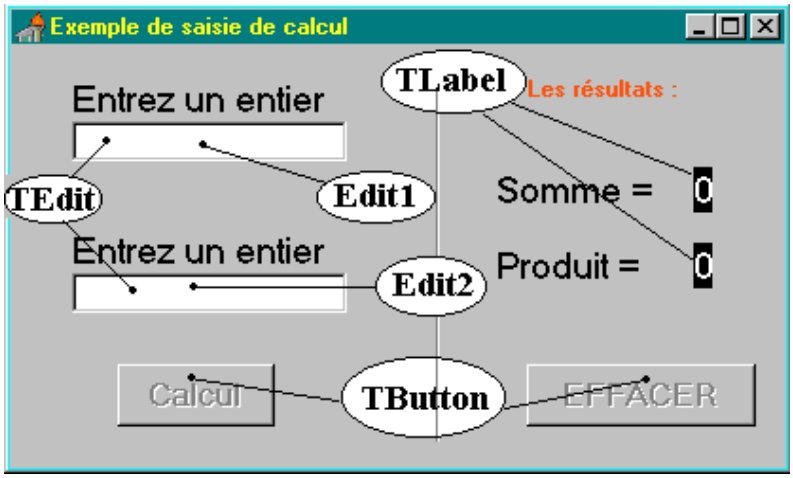
Edit1: TEdit;
Edit2: TEdit;
LabelSomme: TLabel;
LabelProduit: TLabel;

ButtonCalcul: TButton;

Calcul

Buttoneffacer: TButton;

EFFACER



Construction progressive du gestionnaire d'événement Onchange

Nous devons réaliser une synchronisation des entrées dans Edit1 et Edit2 : le déverrouillage (activation) du bouton " ButtonCalcul " ne doit avoir lieu que lorsque Edit1 et Edit2 contiennent des valeurs. Ces deux objets de classe TEdit, sont sensibles à **l'événement OnChange** qui indique que le contenu du champ **text** a été modifié, l'action est indiquée dans le graphe événementiel sous le vocable " *changer* ".

La synchronisation se fera à l'aide de deux drapeaux binaires (des booléens) qui seront levés chacun séparément par les TEdit lors du changement de leur contenu. Le drapeau Som_ok est levé par l'Edit1, le drapeau Prod_ok est levé par l'Edit2.

Implantation : deux champs privés booléens

```
Som_ok , Prod_ok : boolean;
```

Le drapeau Som_ok est levé par l'Edit1 lors du changement du contenu de son champ text, sur l'apparition de l'événement OnChange, il en est de même pour le drapeau Prod_ok et l'Edit2 :

Implantation n°1 du gestionnaire de OnChange :

procedure TForm1.Edit1Change(Sender: TObject); **begin** Som_ok:=true; *// drapeau de Edit1 levé* **end**;	**procedure** TForm1.Edit2Change(Sender: TObject); **begin** Prod_ok:=true; *// drapeau de Edit2 levé* **end**;

Une méthode privée de test vérifiera que les deux drapeaux ont été levés et lancera l'activation du **ButtonCalcul**.

```
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
 if  Prod_ok and Som_ok then
    ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;
```

Nous devons maintenant tester lorsque nous levons un drapeau si l'autre n'est pas déjà levé. Cette opération s'effectue dans les gestionnaires de l'événement OnChange de chacun des Edit1 et Edit2 :

Implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Som_ok:=true; // drapeau de Edit1 levé
  TestEntrees;
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  Prod_ok:=true; // drapeau de Edit2 levé
  TestEntrees;
end;
```

Construction des gestionnaires d'événement Onclick

Nous devons gérer l'événement clic sur le bouton calcul qui doit calculer la somme et le produit, placer les résultats dans les deux Tlabels et activer le Buttoneffacer.

Implantation du gestionnaire de OnClick du ButtonCalcul :

```
procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
    S:=strtoint(Edit1.text); // transtypage : string à integer
    P:=strtoint(Edit2.text); // transtypage : string à integer
    LabelSomme.caption:=inttostr(P+S);
    LabelProduit.caption:=inttostr(P*S);
    Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

Nous codons une méthode privée dont le rôle est de réinitialiser l'interface à son état de départ indiqué dans la table des états initiaux :

Edit1	***Activé***	**procedure** TForm1.RAZTout;
		begin
		Buttoneffacer.Enabled:=false; //le bouton effacer se désactive
		ButtonCalcul.Enabled:=false; //le bouton calcul se désactive
		LabelSomme.caption:='0'; // RAZ valeur somme affichée
		LabelProduit.caption:='0'; // RAZ valeur produit affichée
		Edit1.clear; // message M1
		Edit2.clear; // message M2
		Prod_ok:=false; // RAZ drapeau Edit2
		Som_ok:=false; // RAZ drapeau Edit1
		end;

Edit2	***Activé***	
Buttoncalcul	***Désactivé***	
Buttoneffacer	***désactivé***	

Nous devons gérer l'événement click sur le Buttoneffacer qui doit remettre l'interface à son état initial (par appel à la procédure RAZTout) :

Implantation du gestionnaire de OnClick du Buttoneffacer:

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;
```

Au final, lorsque l'application se lance et que l'interface est céée nous devons positionner tous les objets à l'état initial (nous choisissons de lancer cette initialisation sur l'événement **OnCreate** de création de la fiche):

Implantation du gestionnaire de OnCreate de la fiche Form1:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RAZTout;
end;
```

Si nous essayons notre interface nous constatons que nous avons un problème de sécurité à deux niveaux dans notre saisie :

- Le premier niveau est celui des corrections apportées par l'utilisateur (effacement de chiffres déjà entrés) et la synchronisation avec l'éventuelle vacuité d'au moins un des champs text après une telle modification : en effet l'utilisateur peut effacer tout le contenu d'un " Edit " et lancer alors le calcul, provoquant ainsi des erreurs.

- Le deuxième niveau classique est celui du transtypage incorrect, dans le cas où l'utilisateur commet des fautes de frappe et rentre des données autres que des chiffres (des lettres ou d'autres caractères du clavier).

Améliorations de sécurité de premier niveau par plan d'action

Nous protégeons les calculs dans le logiciel, par un test sur la vacuité du champ text de chaque objet de saisie TEdit. Dans le cas favorable où le champ n'est pas vide, on autorise la saisie ; dans l'autre cas on désactive systématiquement le ButtonCalcul et l'on abaisse le drapeau qui avait été levé lors de l'entrée du premier chiffre, ce qui empêchera toute erreur ultérieure. L'utilisateur comprendra de lui-même que tant qu'il n'y a pas de valeur dans les entrées, le logiciel ne fera rien et on ne passera donc pas au plan d'action suivant (calcul et affichage). Cette amélioration s'effectue dans les gestionnaires d'événement OnChange des deux TEdit.

Implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit1.text<' ' then // champ text non vide ok !
 begin
   Som_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; // bouton désactivé
   Som_ok:=false; // drapeau de Edit1 baissé
 end
```

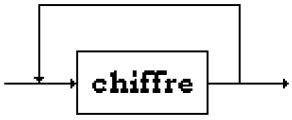
```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit2.text<' ' then // champ text non vide ok !
 begin
   Prod_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
   Prod_ok:=false; // drapeau de Edit2 baissé
 end
```

end;	**end**;

On remarque que les deux codes précédents sont très proches, ils diffèrent par leTEdit et le drapeau auxquels ils s'appliquent. Il est possible de réduire le code redondant en construisant par exemple une méthode privée avec deux paramètres.

Améliorations de sécurité de second niveau par automate de filtrage

Nous pouvons améliorer cet état de la saisie des caractères chiffres en construisant un **analyseur de filtrage** qui ne conserve que les caractères valides tapés dans chaque TEdit. La syntaxe de l'entrée est fournie par le diagramme suivant :



Nous construisons un AEFD (**automate d'états finis**) reconnaissant ces chiffres et il nous servira de système de filtrage des caractères entrés.

L'AEFD de filtrage :

```
procedure Filtrage(entree :string;var resultat:string);
var i:integer;
  saisie :string;
  CarValides:set of char;
begin
 CarValides:=['0'..'9']; // les chiffres seulement
 saisie:=entree;
 if length(saisie)<0 then
 begin
  i:=1;
  while saisie[i] in CarValides do i:=i+1; // le premier caractère non juste
  if not(saisie[i] in CarValides) then delete(saisie,i,1);
 end;
 resultat:=saisie
end;
```

Implantation n°3 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
var sortie :string ;
begin
   Filtrage(Edit1.text,sortie);
   Edit1.text:=sortie;
   Som_ok:=true; // drapeau de Edit1 levé
   TestEntrees;
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
var sortie :string ;
begin
   Filtrage(Edit2.text,sortie);
   Edit2.text:=sortie;
   Prod_ok:=true; // drapeau de Edit2 levé
   TestEntrees;
end;
```

Combinons le niveau 1 et le filtrage effectué lors de la saisie des entrées dans les gestionnaires d'événement OnChange des deux Tedit. Chaque changement du contenu du champ text d'un Edit (comme l'entrée d'un nouveau caractère) provoque le déclenchement de l'événement OnChange qui rejette alors les caractères non valides.

Implantation n°4 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
var sortie :string ;
begin
  Filtrage(Edit1.text,sortie);
  Edit1.text:=sortie;
  if Edit1.text<'' then// champs text non vide ok !
  begin
    Som_ok:=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; // sinon bouton désactivé
    Som_ok:=false; // drapeau de Edit1 baissé
  end
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
var sortie :string ;
begin
  Filtrage(Edit2.text,sortie);
  Edit2.text:=sortie;
  if Edit2.text<'' then // champs text non vide ok !
  begin
    Prod_ok:=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; // sinon bouton désactivé
    Prod_ok:=false; // drapeau de Edit2 baissé
  end
end;
```

En combinant la sécurité du premier et du second niveau dans le gestionnaire d'événement OnChange nous obtenons un niveau acceptable de sécurisation de notre logiciel grâce à son interface. Il nous manque encore une protection sur les débordements de calcul (dépassement de l'intervalle sur les integer qui ne provoquent pas d'incidents mais induisent des résultats erronés) pour assurer une sécurité optimale de fiabilité.
Pour terminer la réalisation de cet exemple, nous pouvions aussi procéder à un autre genre de protection sans avoir à écrire un analyseur de filtrage (qui était dans ce cas un AEFD), en utilisant directement les facilités de protection fournies par les exceptions.

Améliorations de sécurité de second niveau par exceptions

Dans cette éventualité, l'on déporte le problème de la protection non plus sur le séquencement des plans d'actions mais au cœur même de l'action en protégeant directement le code où l'erreur se produit par un gestionnaire d'exception. Afin de présenter un traitement complet de l'exemple nous anticipons légèrement sur le chapitre sur la programmation défensive ; le lecteur pourra donc en première lecture sauter ce paragraphe s'il le désire et y revenir plus tard.

On fait exécuter le programme en provoquant volontairement l'erreur (on entre des lettres au lieu de chiffres) ; le logiciel signale la levée de l'exception EconvertError et nous programmons le gestionnaire associé à cette levée d'exception. Elle apparaît lorsque la fonction StrtoInt essaye de transtyper le champ text de l'Edit et échoue ; c'est donc cette ligne de code qui doit être protégée :

```
try
    S:= StrtoInt(edit1.text);
 except on EconvertError do
 begin
   Edit1.text:='0';
   S:=0
 end
end;
```

Nous avons décidé ici de mettre 0 dans le champ text de l'Edit1 et de forcer la valeur de la variable de calcul S à 0. Ce traitement est identique pour la variable P à partir du champ text de l'Edit2. Ces deux traitements de protection sont effectués lors du click sur ButtonCalcul (traitement de l'erreur après saisie).

```
procedure TForm1.ButtonCalculClick(Sender: TObject);
var S,P:integer;
begin
 try
    S:= StrtoInt(edit1.text);
  except on EconvertError do
  begin
    Edit1.text:='0';
    S:=0
  end
 end;
 try
    P:= StrtoInt(edit2.text);
  except on EconvertError do
  begin
    Edit2.text:='0';
    P:=0
  end
 end;
 LabelSomme.caption:=inttostr(P+S);
 LabelProduit.caption:=inttostr(P*S);
 Buttoneffacer.enabled:=true // le bouton effacer est activé
end;{ButtonCalculClick}
```

Regroupement du code

Reprenons le code du premier niveau par plan d'action en le regroupant dans la méthode privée Autorise possédant deux paramètres, le premier est la référence d'un des deux TEdit, le second le drapeau booléen associé :

```
procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
 if Ed.text<' ' then // champ text non vide ok !
 begin
   flag :=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
    flag:=false; // drapeau de Ed baissé
 end
end;
```

Nouvelle implantation n°2 du gestionnaire de OnChange :

procedure TForm1.Edit1Change(Sender: TObject); **begin** Autorise (Edit1 , Som_ok); **end**;	**procedure** TForm1.Edit2Change(Sender: TObject); **begin** Autorise (Edit2 , Prod_ok); **end**;

Code final où tout est regroupé dans la classe TForm1

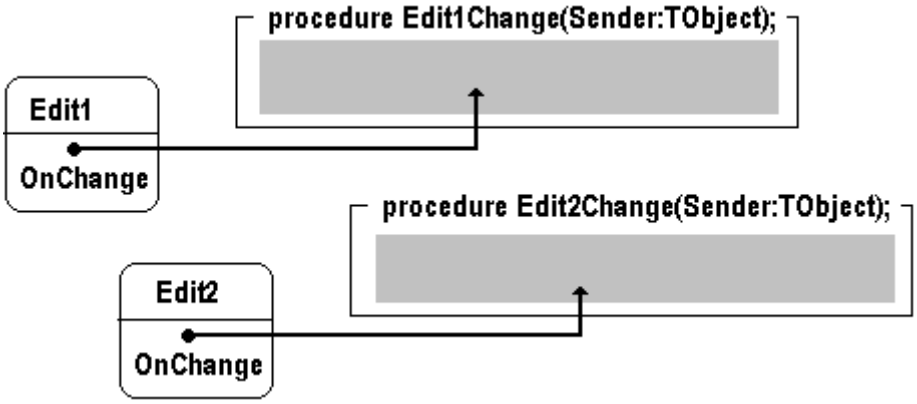
```
unit UFcalcul;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls, ExtCtrls;

type
 TForm1= class ( TForm )
    Edit1: TEdit;
    Edit2: TEdit;
    ButtonCalcul: TButton;
    LabelSomme: TLabel;
    LabelProduit: TLabel;
    Buttoneffacer: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure Edit2Change(Sender: TObject);
    procedure ButtonCalculClick(Sender: TObject);
    procedure ButtoneffacerClick(Sender: TObject);
  private   { Déclarations privées }
      Som_ok , Prod_ok : boolean;
      procedure TestEntrees;
      procedure RAZTout;
      procedure Autorise ( Ed : TEdit ; var flag : boolean );
  public   { Déclarations publiques }
  end;

implementation
{ --------------- Méthodes privées --------------------}
```

```
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
  if  Prod_ok and Som_ok then
     ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;

procedure TForm1.RAZTout;
begin
    Buttoneffacer.Enabled:=false; //le bouton effacer se désactive
    ButtonCalcul.Enabled:=false; //le bouton calcul se désactive
    LabelSomme.caption:='0'; // RAZ valeur somme affichée
    LabelProduit.caption:='0'; // RAZ valeur produit affichée
    Edit1.clear; // message M1
    Edit2.clear; // message M2
    Prod_ok:=false; // RAZ drapeau Edit2
    Som_ok:=false; // RAZ drapeau Edit1
end;

procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
  if Ed.text<>' ' then // champ text non vide ok !
  begin
    flag :=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; //bouton désactivé
     flag:=false; // drapeau de Ed baissé
  end
end;

{ ------------------- Gestionnaires d'événements -------------------}
procedure TForm1.FormCreate(Sender: TObject);
begin
   RAZTout;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
   Autorise ( Edit1 , Som_ok );
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
   Autorise ( Edit2 , Prod_ok );
end;

procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
     S:=strtoint(Edit1.text); // transtypage : string à integer
     P:=strtoint(Edit2.text); // transtypage : string à integer
     LabelSomme.caption:=inttostr(P+S);
     LabelProduit.caption:=inttostr(P*S);
     Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;

end.
```

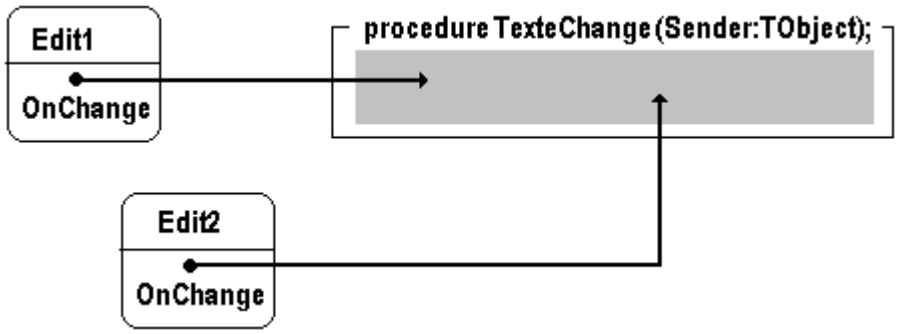
Un gestionnaire d'événement centralisé grâce au :
- **polymorphisme d'objet**
- **pointeur de méthode**

Dans le code précédent, chaque Edit possède son propre gestionnaire de l'événement OnChange :



procedure TForm1.Edit1Change(Sender: TObject); **begin** Autorise (Edit1 , Som_ok); **end**;	**procedure** TForm1.Edit2Change(Sender: TObject); **begin** Autorise (Edit2 , Prod_ok); **end**;

On peut aussi mettre en place un gestionnaire unique de l'événement OnChange, puis de le lier à chaque zone de saisie Edit1 et Edit2 (souvenons-nous qu'un champ d'événement est un pointeur de méthode) :



Nous définissons d'abord une méthode public par exemple, qui jouera le rôle de gestionnaire centralisé d'événement, nous la nommons TexteChange.

Cette méthode pour être considérée comme un gestionnaire de l'événement OnChange, doit obligatoirement être compatible avec le type de l'événement Onchange. Une consultation de la documentation Delphi nous indique que :

property OnChange: TNotifyEvent;

L'événement OnChange est du même type que l'événement OnClick (TnotifyEvent = **procedure**(Sender:Tobject) **of object**), donc notre gestionnaire centralisé TexteChange doit avoir l'en-tête suivante :

procedure TexteChange (Sender : Tobject);

Le paramètre Sender est la référence de l'objet qui appelle la méthode qui est passé automatiquement lors de l'exécution, en l'occurrence ici lorsque Edit1 appellera ce gestionnaire c'est la référence de Edit1 qui sera passée comme paramètre, de même lorsqu'il s'agira d'un appel de Edit2.

Implantation du gestionnaire centralisé

```
procedure TForm1.TexteChange(Sender: TObject);
begin
  if  Sender is TEdit then
  begin
     if (Sender as TEdit )=Edit1 then
        Autorise ( (Sender as TEdit ) , Som_ok )
     else
        Autorise ( (Sender as TEdit ) , Prod_ok );
   end
end;
```

On teste si l'émetteur Sender est bien un TEdit,

polymorphisme d'objet :

Si l'émetteur est Edit1 on le **transtype** en TEdit pour pouvoir le passer en premier paramètre à la méthode Autorise **sinon** c'est Edit2 et l'on fait la même opération.

On lie maintenant ce gestionnaire à chacun des champs OnChange de chaque TEdit dès la création de la fiche :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RAZTout;
  Edit1.OnChange := TexteChange ;
  Edit2.OnChange := TexteChange ;
end;
```

pointeur de méthode :

chaque champ Onchange pointe vers le même gestionnaire (méthode)

```
TForm1= class ( TForm )
   Edit1: TEdit;      …
   procedure FormCreate(Sender: TObject);
   procedure ButtonCalculClick(Sender: TObject);
   procedure ButtoneffacerClick(Sender: TObject);
 private   { Déclarations privées }
    Som_ok , Prod_ok :
    procedure TestEntrees;
    procedure RAZTout;
    procedure Autorise ( Ed : TEdit ; var flag : boolean );
 public   { Déclarations publiques }
   procedure TexteChange(Sender: TObject);
 end;
```

Le gestionnaire centralisé est déclaré ici, puis il est implémenté à la section **implementation** de la **unit**.

Ce logiciel d'interface simplifiée a permis de mettre en œuvre tous les concepts de base d'une interface à l'exception des temps d'attente qui ne sont pas pertinents dans cet exemple :

> les objets d'entrée-sortie sont fournis par le RAD,
> le pilotage a été réalisé à l'aide du graphe événementiel,
> l'enchaînement a été implanté à travers la synchronisation dans le graphe événementiel,
> une partie de la sécurité a été obtenue en décomposant l'interaction homme-logiciel en deux plans d'action : le plan de saisie et le plan de calcul-affichage.

Corrections des exercices sur les BD (2,3,4,5)

Rappelons que vous pourrez développer avec delphi des logiciels d'accès à une BD Access uniquement si vous possédez les deux logiciels suivants :

- Le SGBD-R Microsoft Access
- Une version de Delphi contenant le BDE (version professionnelle, version développeur, version entreprise, version….) Les versions standard et gratuites dites personnelles, ne contiennent pas de composants de Base de Données et vous ne pourrez pas développer ces exercices.

Le minimum requis pour faire fonctionner les exercices ci-après est :

- Delphi professionnel 5
- Access 97
- Windows Xp

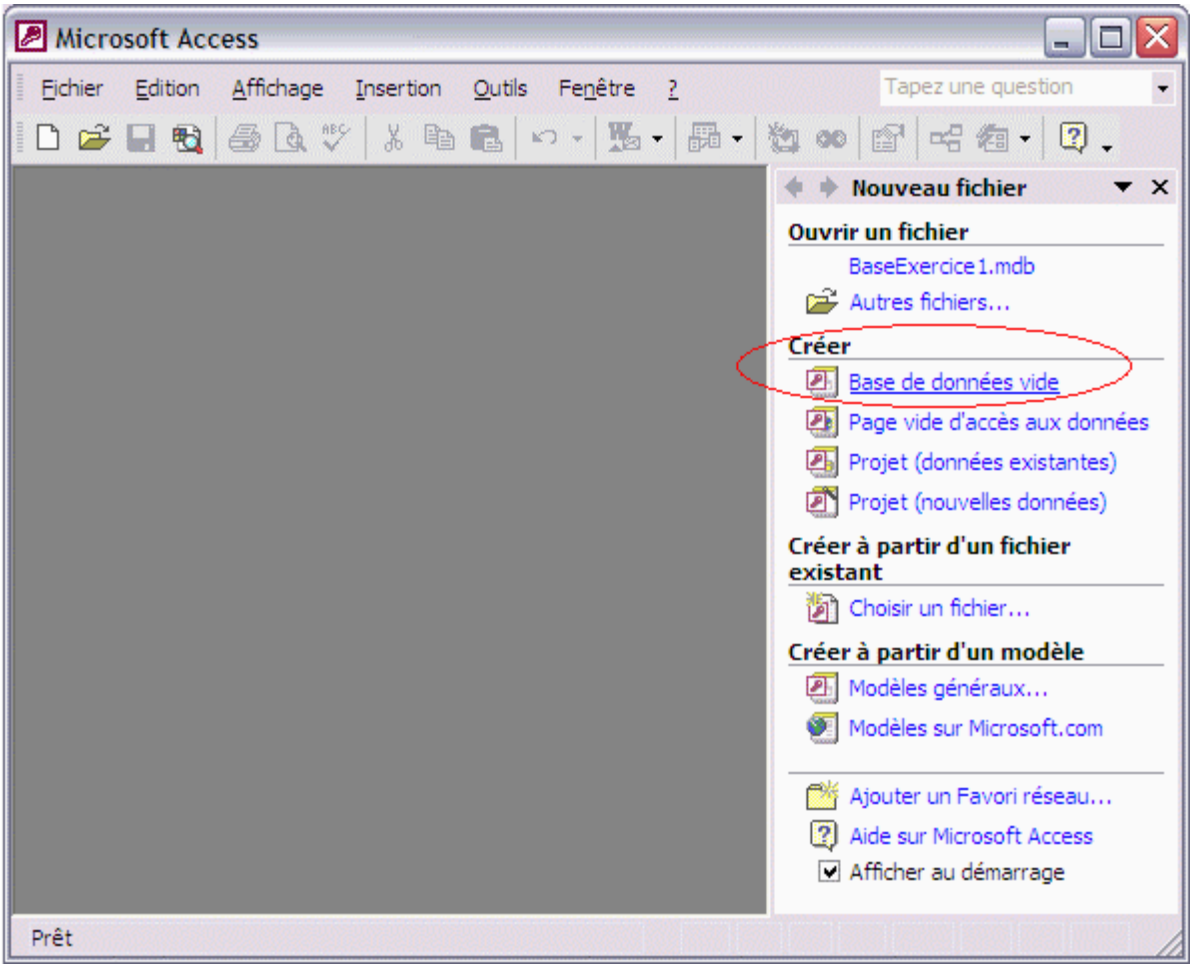
(ils ont aussi été testés avec Delphi 7 professionnel et Access 2002 sous Xp)

Démarche de présentation des solutions des exercices :

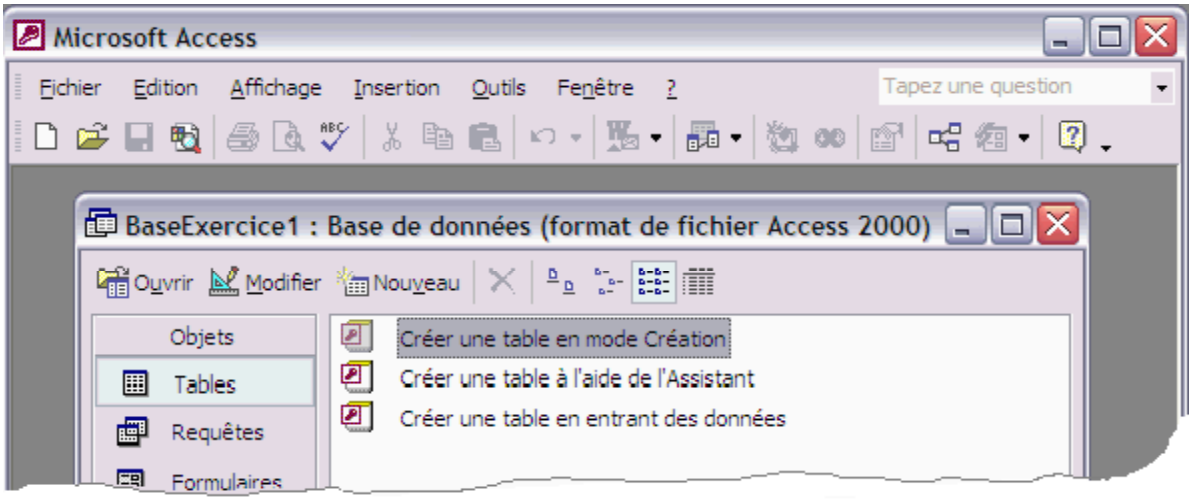
- Nous décrivons sous forme synoptique comment créer la base de données BD.
- Nous montrons ensuite comment gérer des alias de BD à travers le BDE de Borland
- Puis pour chacun des 4 exercices :
  - Nous donnons la construction de l'interface de l'exercice avec Delphi.
  - Nous donnons le code source de l'exercice en Delphi.

Décrivons d'abord pas à pas la création de la BD avec le SGBD-R Access (les images présentées correspondent ici à la version Access 2002).

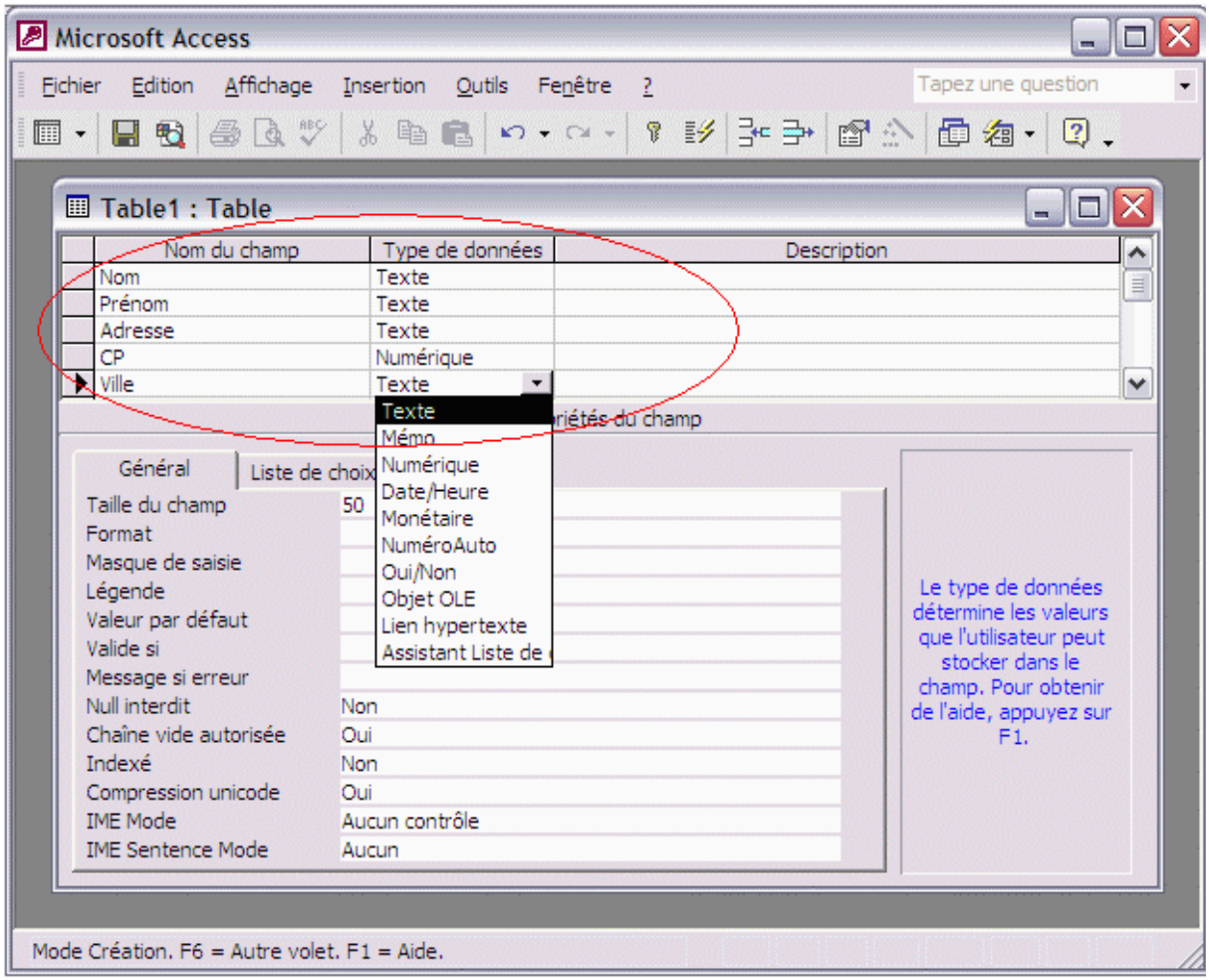
1°) créer une base de données (vide au départ) :



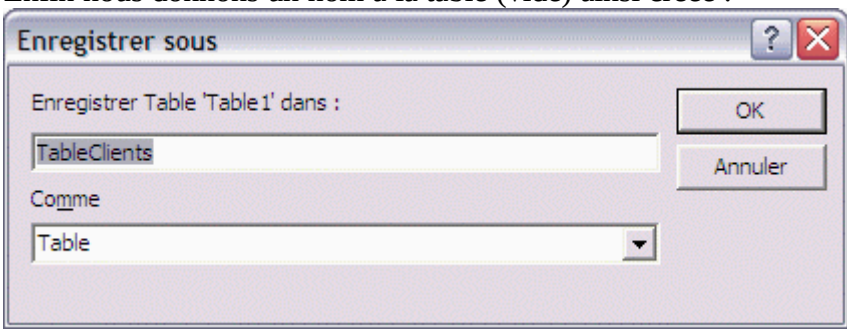
Il faut créer la structure de la table TableClients :



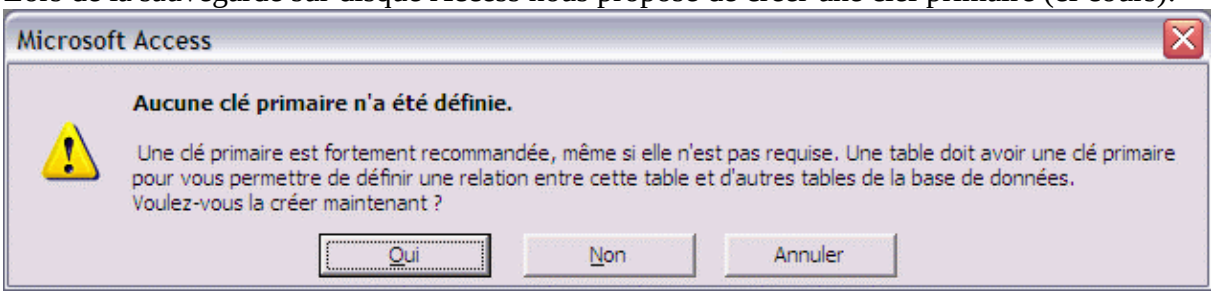
Nous devons nommer et typer les champs de la table :



Enfin nous donnons un nom à la table (vide) ainsi créée :

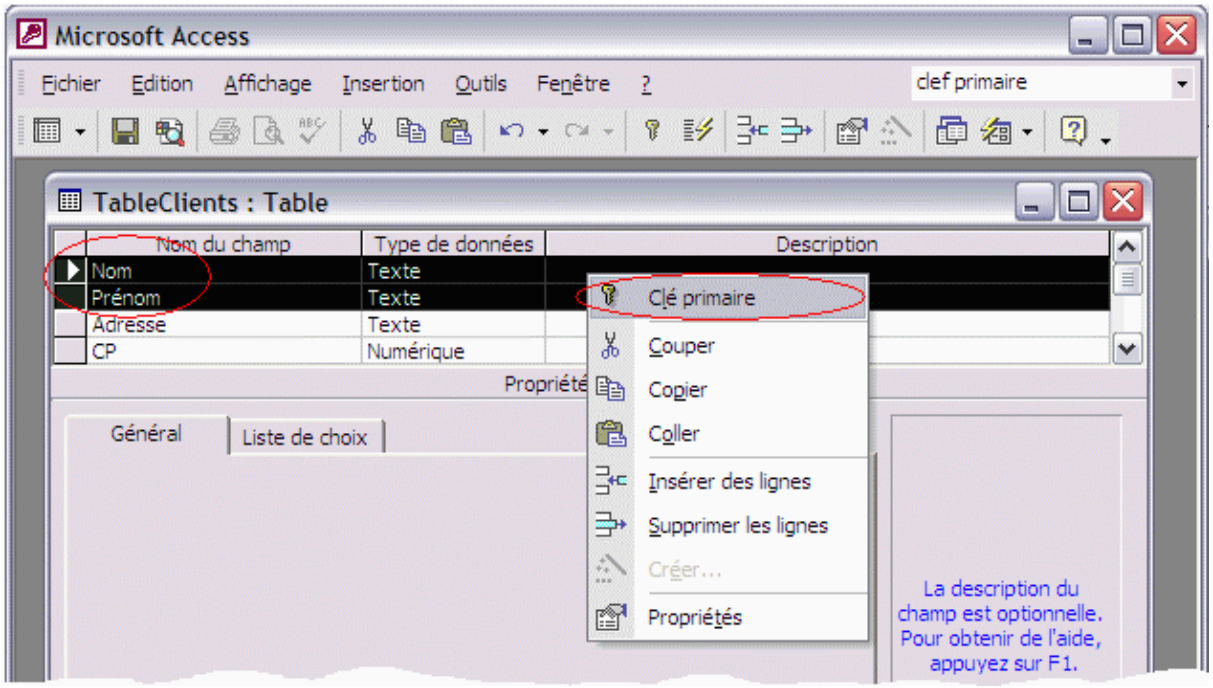


Lors de la sauvegarde sur disque Access nous propose de créer une clef primaire (cf cours):

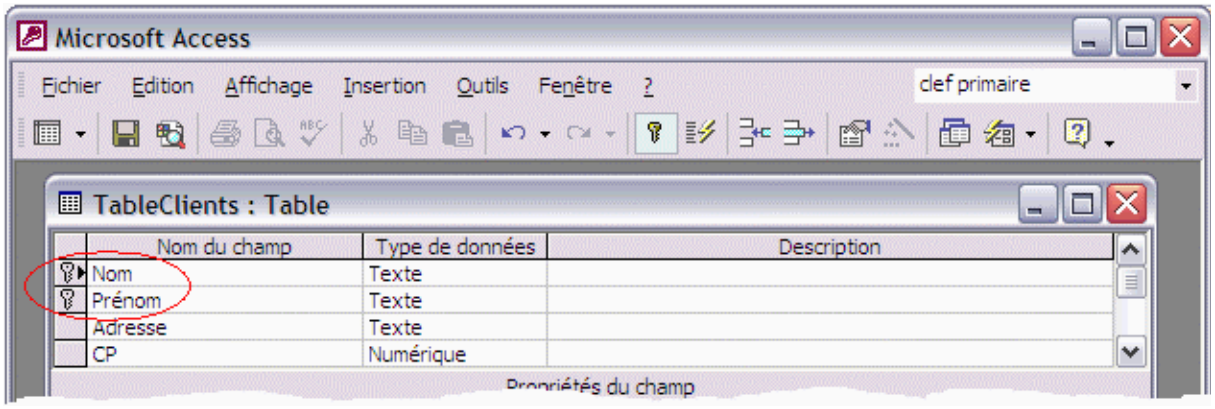


Nous acceptons de définir notre clef primaire en répondant oui à cette invite. Nous définissons une clef primaire possédant 2 champs : le **nom** et le **prénom**. Nous sélectionnons les deux

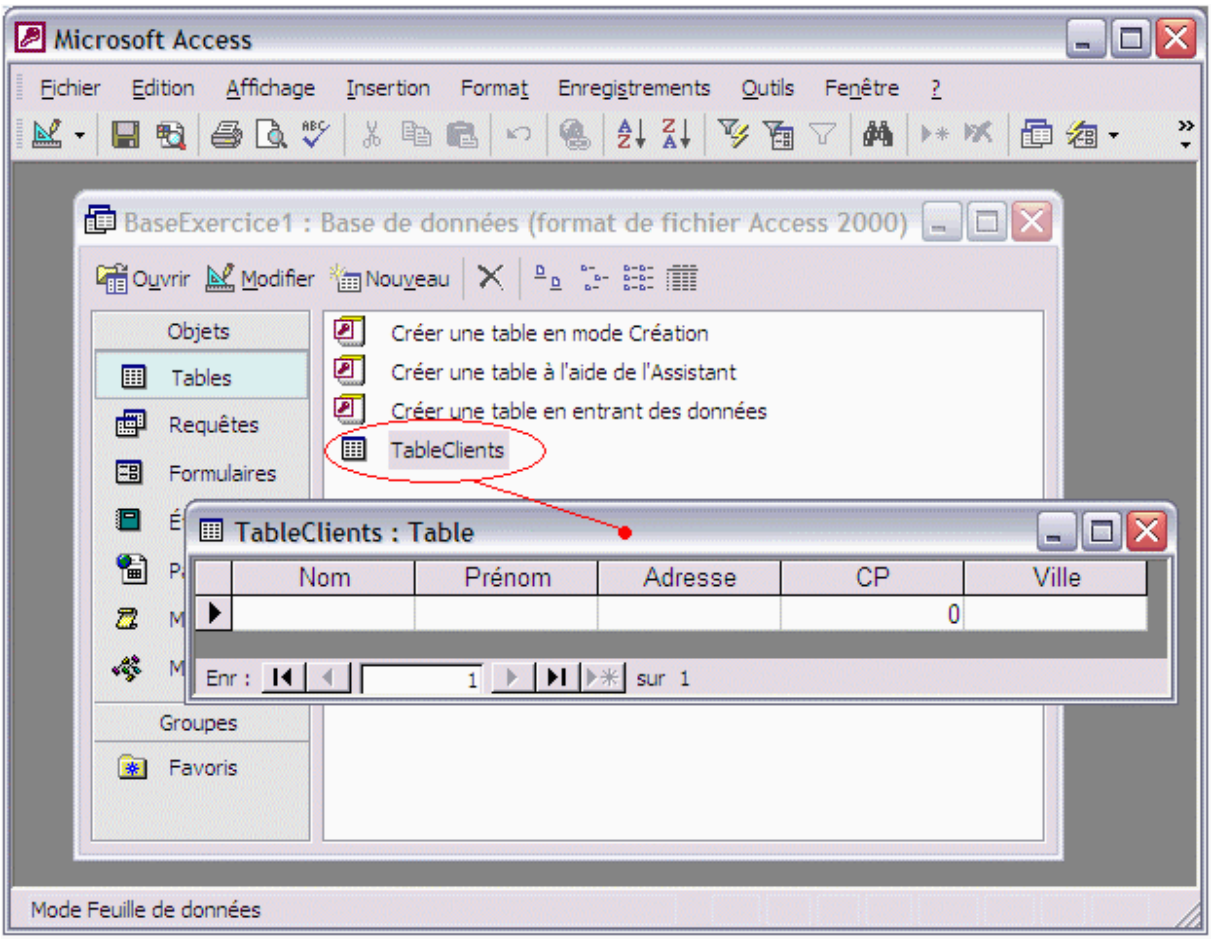
lignes nom et prénom dans l'assistant de création de la table et avec le bouton droit de souris, nous indiquons que ces deux champs (les 2 lignes sélectionnées) constituent la clef primaire de notre relation :



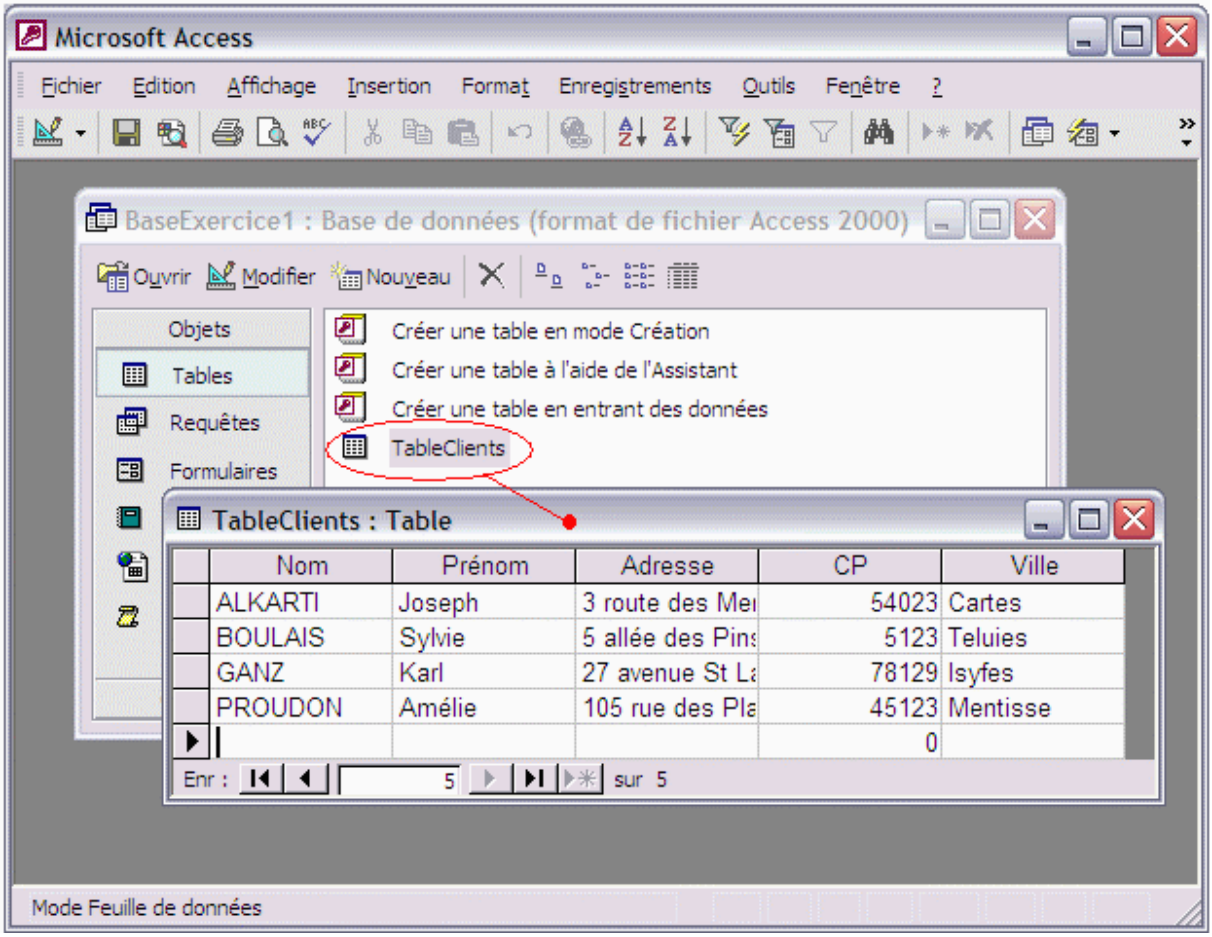
Access nous signale par une petite icône de clef les champs composant la clef primaire de la relation :



La structuration de la table TableClients est enfin terminée et l'assistant du SGBD Access nous fournit la table vide prête à être saisie :



Voici affiché par l'interface du SGBD Access, notre table une fois remplie manuellement :



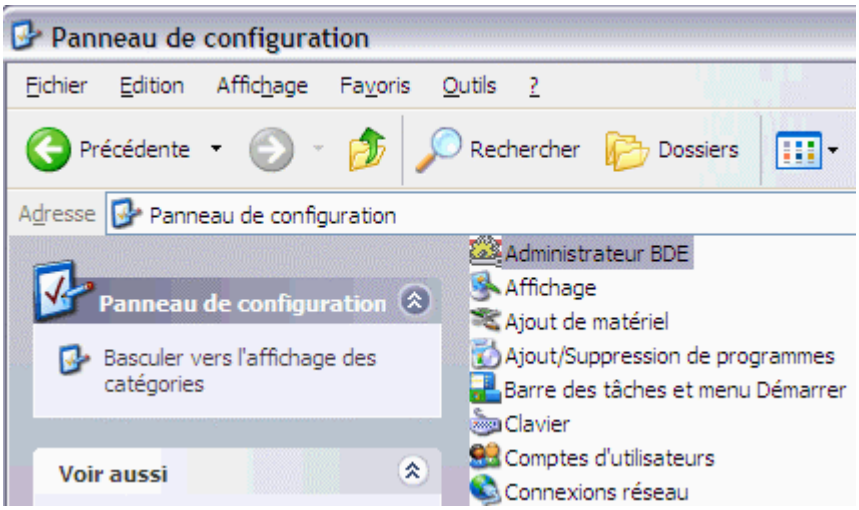
Cette BD est sauvegardée sous le nom : **BaseExercice1.mdb**

TableClients : Table

Nom	Prénom	Adresse	CP	Ville
ALKARTI	Joseph	3 route des Merles	54023	Cartes
BOULAIS	Sylvie	5 allée des Pins	05123	Teluies
GANZ	Karl	27 avenue St Laurent	78129	Isyfes
PROUDON	Amélie	105 rue des Platanes	45123	Mentisse

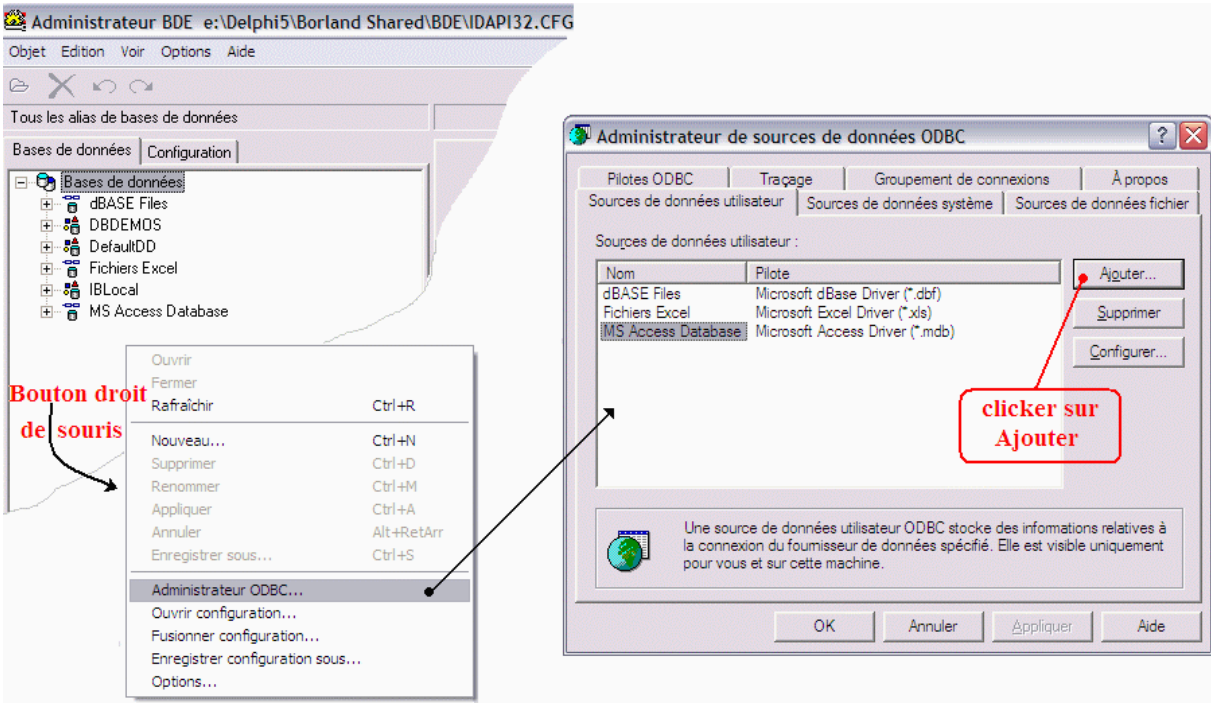
2°) Gestion des alias de BD à travers le BDE

Nous nous servons du moteur de base de données appelé le BDE de Borland pour accéder à notre base Access à partir d'un programme Delphi. Nous utilisons l'outil d'administration du BDE nommé « Administrateur BDE » mis à disposition par Delphi dans le dossier « Panneau de configuration ». Il est alors possible de paramétrer (établir la connexion physique avec la BD) le BDE directement grâce à l'administrateur du BDE fournit avec Delphi :



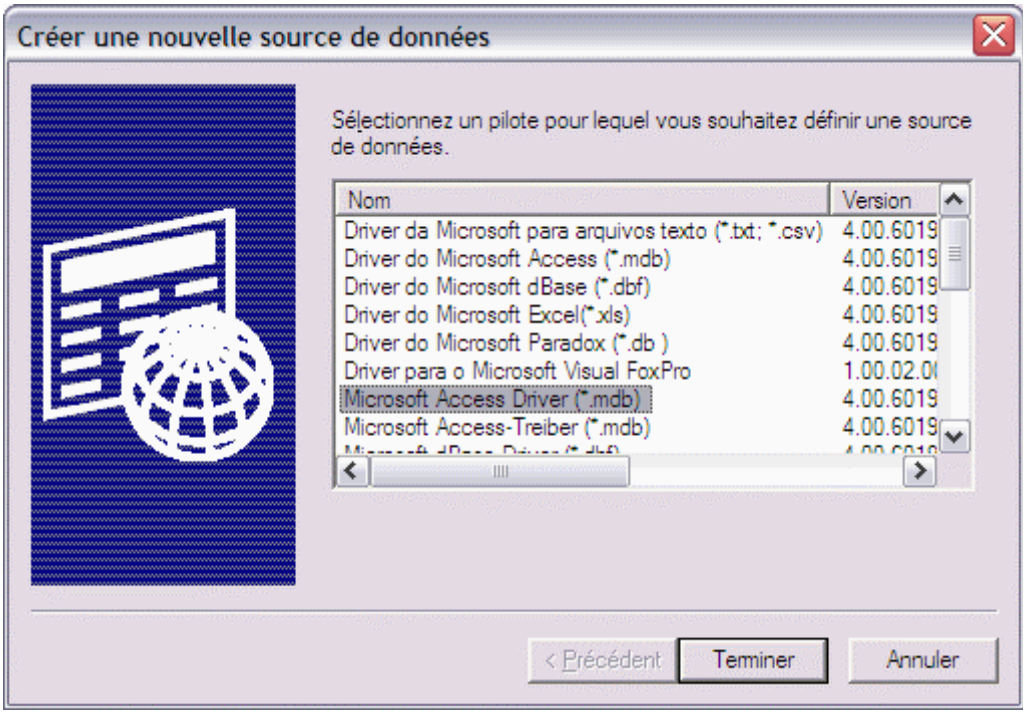
Dans le cours les exemples ont utilisé un pilote dit natif "MSACCESS" fourni par Borland et paramétrant le BDE à partir du composant TDatabase. Dans les exercices, nous allons utiliser un autre genre de pilote de type ODBC (l'Open Data Base Connectivity est un standard d'accès aux BD) de Microsoft à la place du pilote natif fourni.

Nous appelons l'administrateur du BDE à partir du panneau de configuration (cf. chap 7.5) , puis nous demandons à créer un nouvel alias de connexion physique par pilote ODBC :

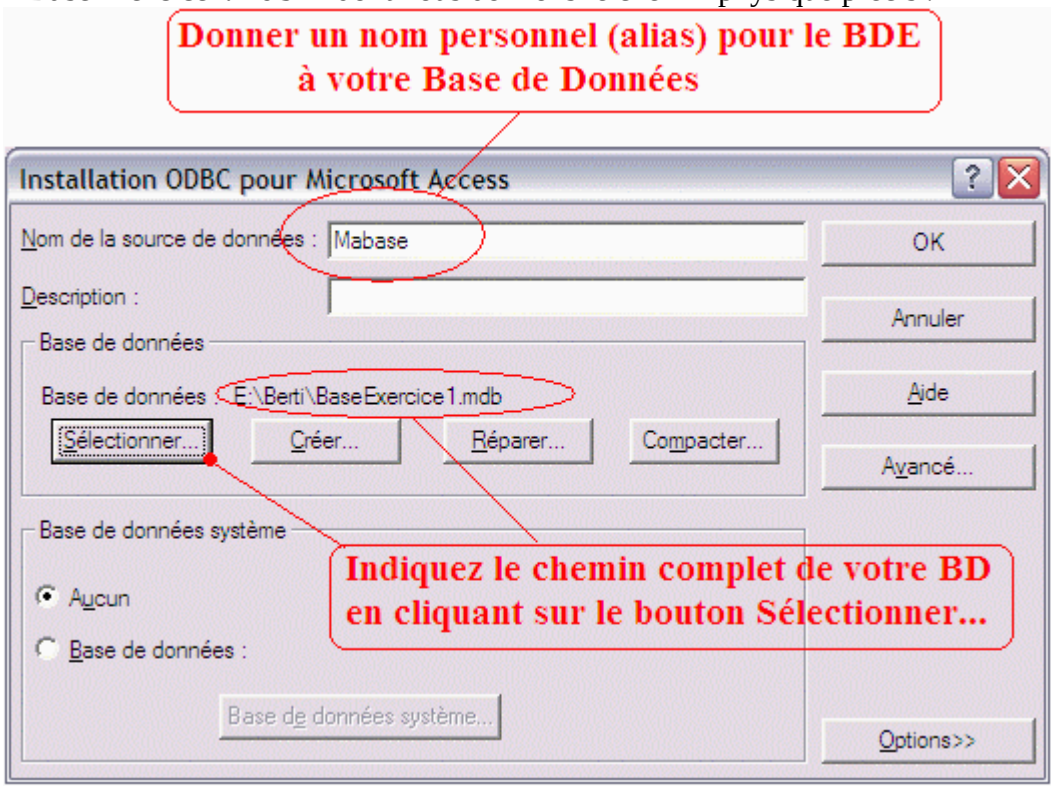


Nous ajoutons dans l'onglet « Sources de données utilisateur » notre BD « BaseExercice1.mdb »

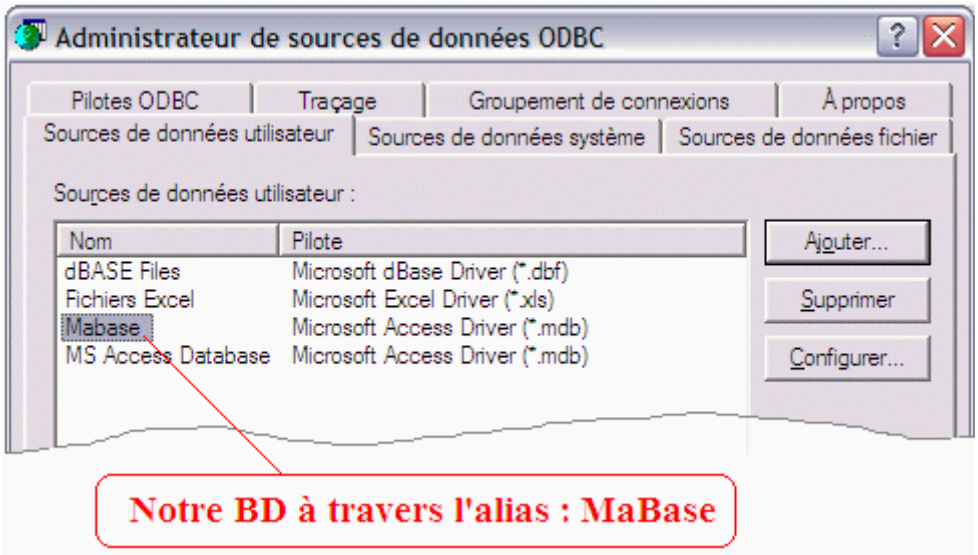
a) en sélectionnant le pilote ODBC fourni par Microsoft dans Access, dénommé « **Microsoft Access Driver** » :



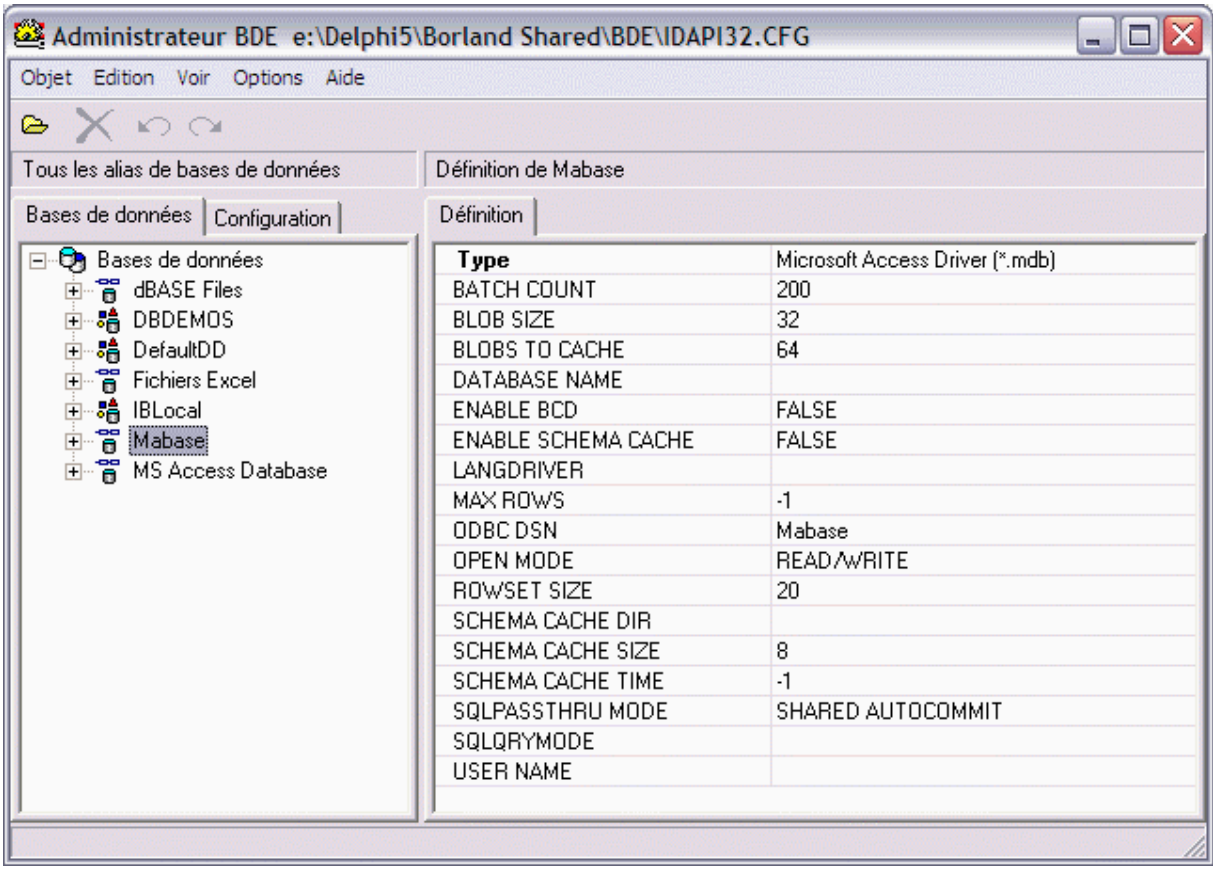
b) en reliant ce pilote avec un nom d'alias que nous choisissons (ici : **MaBase**) et notre BD « **BaseExercice1.mdb** » dont nous donnons le chemin physique précis :



La fenêtre d'administration de sources de données ODBC a enregistré notre nouvelle Base avec son nom « MaBase » :



L'administrateur du BDE contient et affiche maintenant une BD nouvellement accessible qui se dénomme « MaBase » :

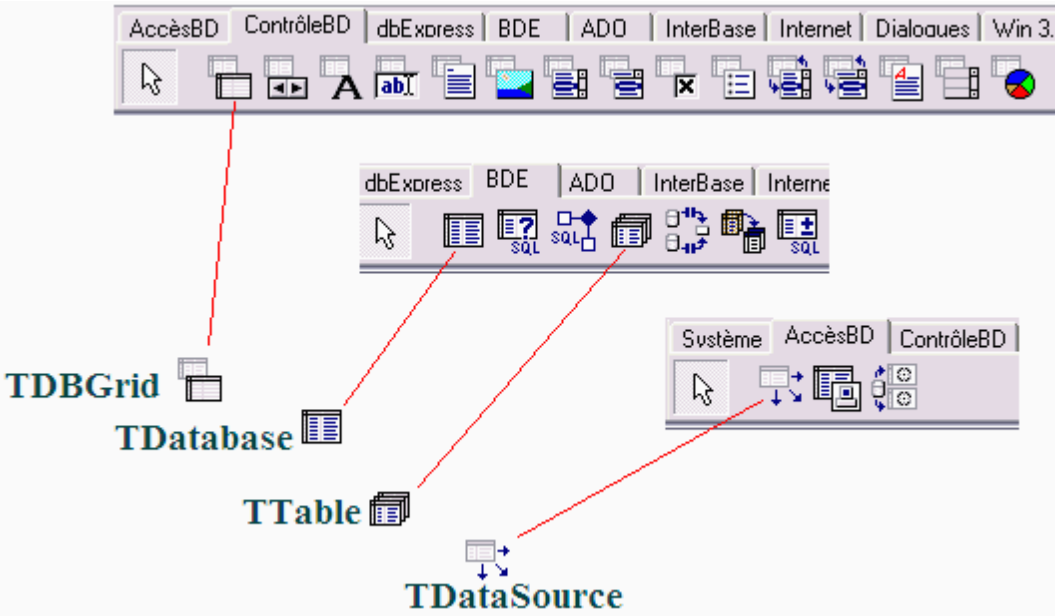


Ces opérations terminées, la BD « **BaseExercice1.mdb** » est dès lors utilisable sous le nom **MaBase** par des programmes Delphi en consultation ou en mise à jour.

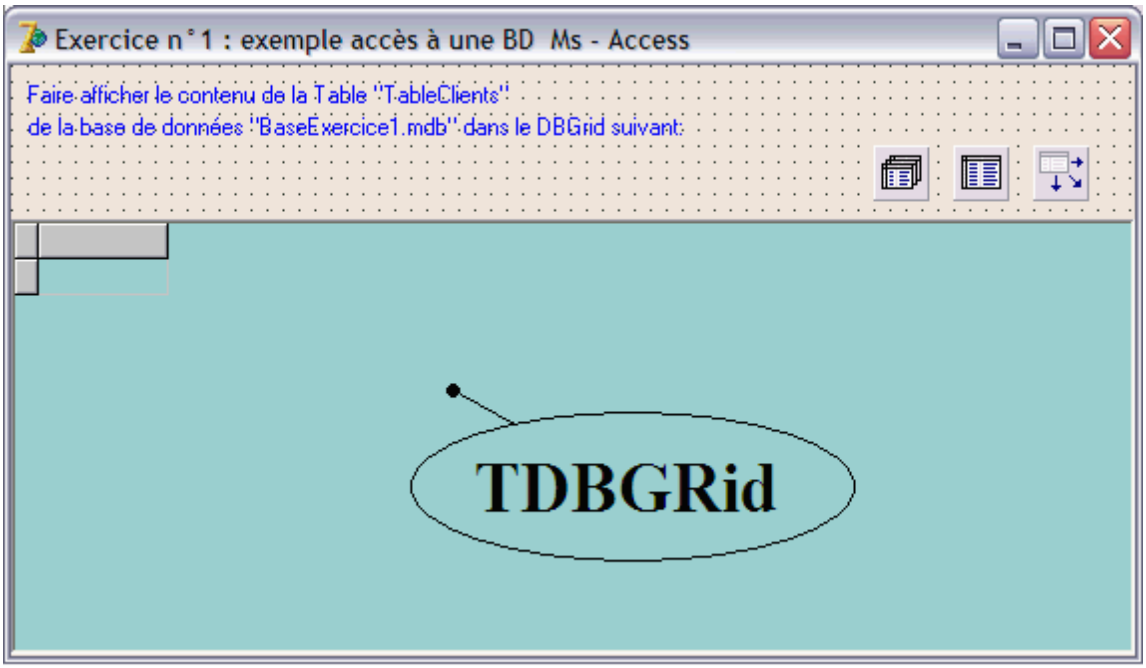
Ex-2 : construire une IHM Delphi - consultation

3°) construction de l'interface de consultation de la BD en Delphi

La démarche complète de création de l'interface avec Delphi se trouve au paragraphe 5.3.2 du chapitre 7.5 sur l'utilisation des bases de données.

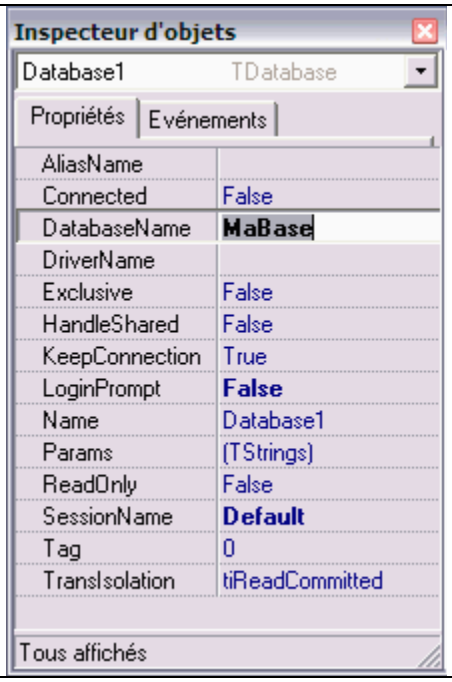


Parmi les 4 composants précédents un seul est un composant visuel le **TDBGrid** :



Pour préparer le programme, il nous faut relier le **TDataBase** à la base de données physique (cela est fait à travers le BDE) :

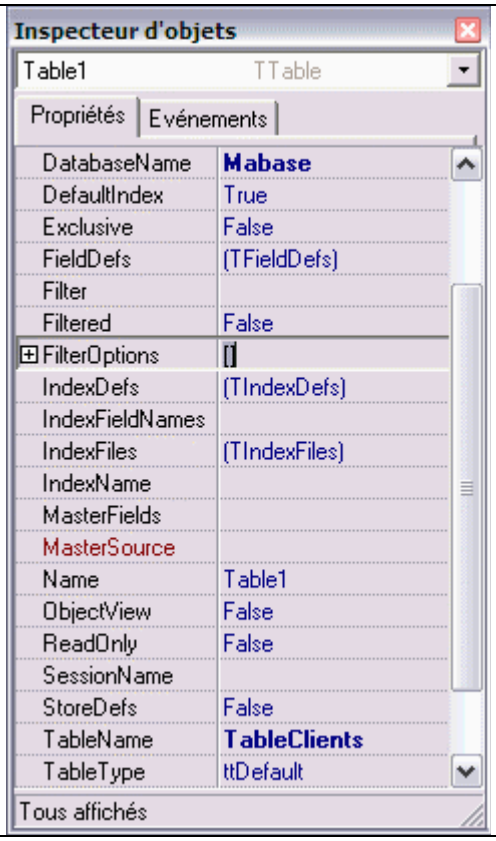
Pour effectuer cette liaison, il nous suffit de donner dans le champ ***DataBaseName*** du TDataBase, le nom de l'alias que nous avons utilisé dans le BDE : « **MaBase** »



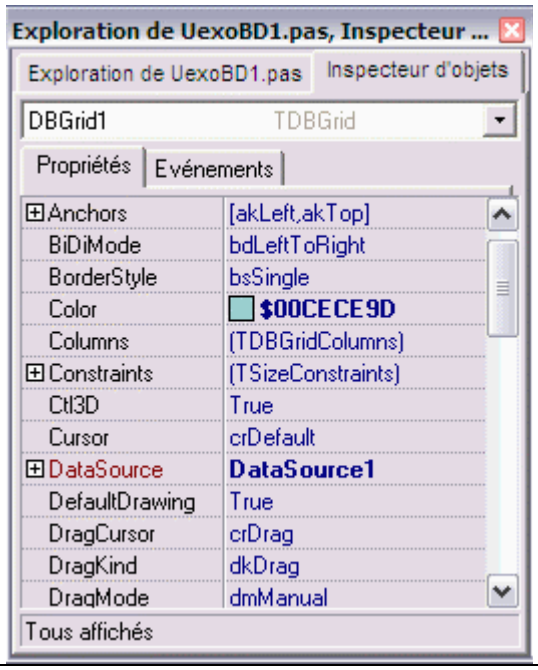
Comme nous voulons afficher la table « TableClients », il nous faut utiliser le composant **TTable** qui permet d'accéder aux données d'une table de base de données en utilisant le BDE ; Il faut indiquer au **TTable** quelle est le nom de la BD à laquelle il doit être connecté et la table de cette base à laquelle il doit être relié :

Le champ ***DataBaseName*** du TTable contient le nom (alias) de la BD à laquelle il est connecté.

Le champ ***TableName*** du TTable contient le nom de la table à laquelle il est relié



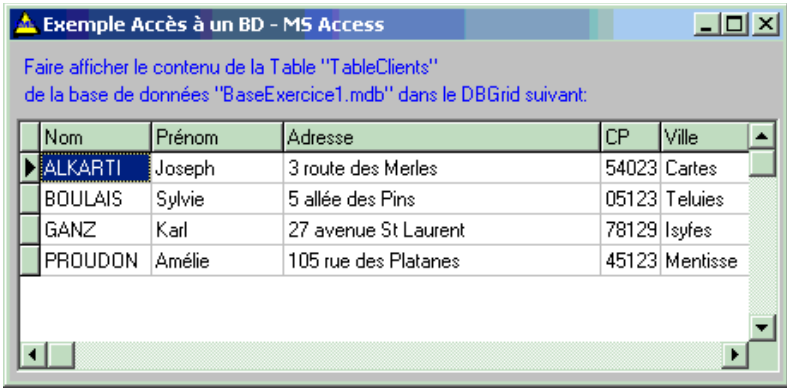
Le **TDataSource** est relié à la fois à la table à afficher à travers le **TTable** et au composant visuel d'affichage le **TDBGrid**:

Le TDataSource	Le TDBGrid
	

4°) Code source delphi pour l'affichage du contenu de la table de la BD

```
unit uFExercice1;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, Grids, DBGrids, StdCtrls, Mask, DBTables, Db;

 type
  TFExercice1 = class(TForm)
  DBGrid1: TDBGrid;
  Table1: TTable;
  DataSource1: TDataSource;
  Database1: TDatabase;
  Label1: TLabel;
  Label2: TLabel;
  procedure FormCreate(Sender: TObject);
 private
 { Déclarations privées }
 public
 { Déclarations publiques }
end;

 var
  FExercice1: TFExercice1;
  App_Path:string;

implementation
```

```
{$R *.DFM}
```




```
procedure TFExercice1.FormCreate(Sender: TObject);
 begin
   DataBase1.connected:=true;  //connexion sur la base de données
  //(évidemment il faut d'abord paramétrer
  //correctement le composant DataBase1)
   Table1.open; //Ouverture de la table attachée au composant Table1
                //(il faut avoir bien paramétré le composants Table1auparavant
 end;

end.
```

Nous remarquons que le code est très court; ce sont les composant qui font tout le travail parce qu'ils ont été paramétrés pendant la conception.

Autre solution de code : le paramétrage des composants lors de la création de la fiche

Dans cette deuxième éventualité, le **TDataBase** et le **TTable** ne sont pas paramétrés lors du dépôt visuel comme dans les schémas présentés ci-haut lors de la conception grâce à l'inspecteur d'objets.

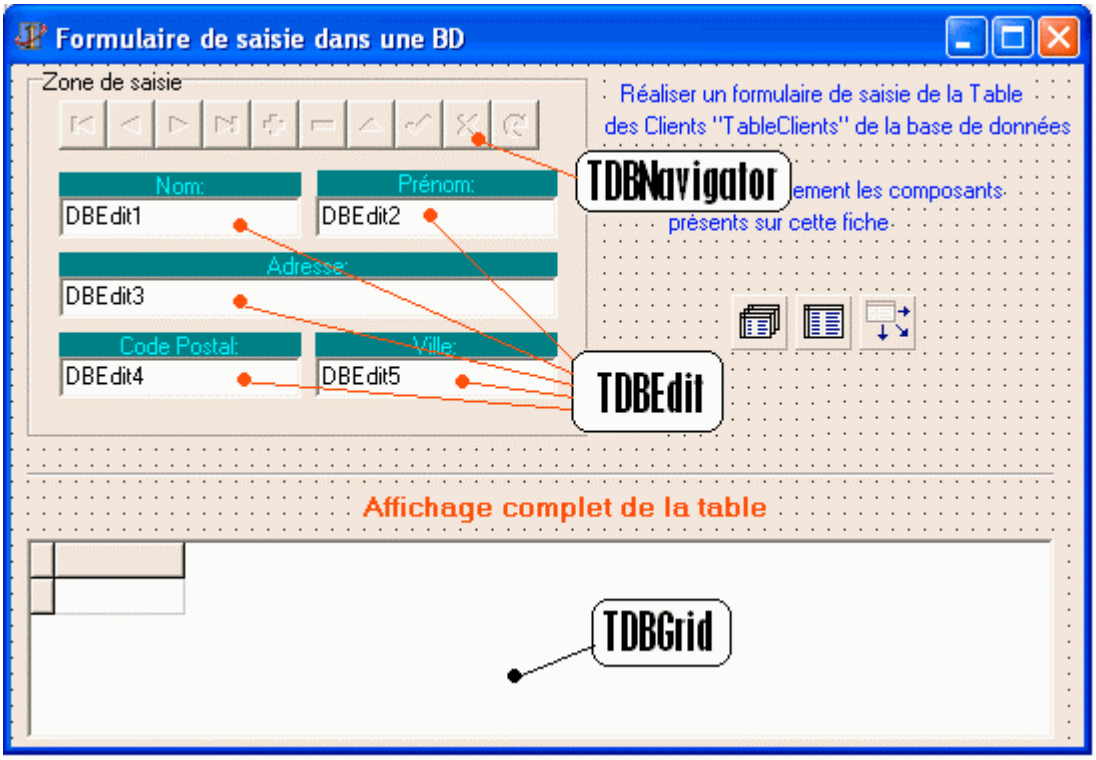
Ils sont paramétrés directement par programmation, pendant l'exécution et lors de la survenue de l'événement de création de la fiche :

```
procedure  TFExercice1.FormCreate( Sender:  TObject) ;
  begin
    DataBase1.DatabaseName := 'MaBase';
    DataBase1.connected := true ;  //connexion sur la base de données

    Table1.DatabaseName := 'MaBase';
    Table1.TableName := 'TableClients';
    Table1.open ;  //Ouverture de la table attachée au composant Table1
   end;
```

Ceci montre que ces composants peuvent, pendant l'exécution être paramétrés à nouveau sur une autre BD existante et jouer leur rôle d'accès à cette nouvelle base.

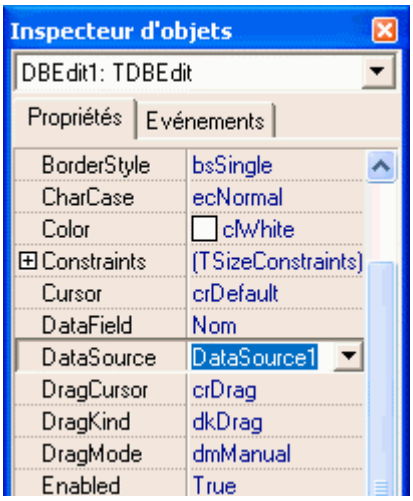
Pour les exercices suivants, nous ne paramétrerons pas le **TDataBase** et le **TTable** lors de la conception, mais par programme comme indiqué dans ce paragraphe.

3°) construction de l'interface de navigation dans la BD en Delphi



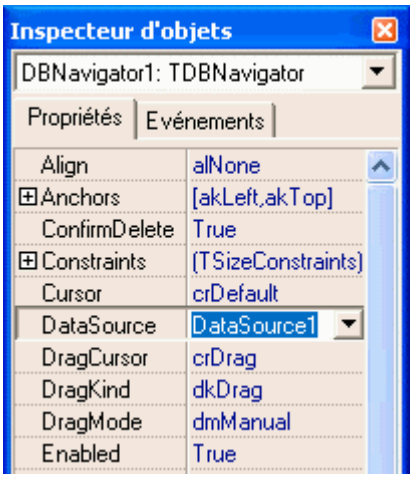
Le **TDataSource** est relié à la fois à la table à afficher à travers le **TTable** et au composant visuel d'affichage le **TDBGrid.** Le **TDataBase** est relié à la base de données physique comme dans l'exercice précédent.

Les quatre **TDBEdit** représentent des contrôles de saisie pouvant afficher et modifier les valeurs d'un champ de la BD. Chacun d'eux est relié par sa propriété DataSource au composant **TDataSource** (nommé DataSource1) :



etc …

Le **TDBNavigator** est un contrôle utilisé pour se déplacer dans une BD afin d'effectuer des actions sur les données(suppression, modification, insertion,…), il est relié par sa propriété DataSource au même composant **TDataSource** (nommé DataSource1) que les **TDBEdit** et le **TDBGrid** :



4°) Code source delphi pour la navigation dans la table de la BD

```
unit  UBDExo2 ;

 interface

 uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Grids, DBGrids, ExtCtrls, Mask, DBCtrls, Db, DBTables ;

 type
   TForm1  =  class (TForm)
  DBGrid1 :  TDBGrid ;
   Label1 :  TLabel ;
   Label2 :  TLabel ;
   Label4 :  TLabel ;
   Label5 :  TLabel ;
   Label6 :  TLabel ;
   GroupBox1 :  TGroupBox ;
   DBNavigator1 :  TDBNavigator ;
   DBEdit1 :  TDBEdit ;
   DBEdit2 :  TDBEdit ;
   DBEdit3 :  TDBEdit ;
   DBEdit4 :  TDBEdit ;
   DBEdit5 :  TDBEdit ;
   Label7 :  TLabel ;
   Label8 :  TLabel ;
   Label9 :  TLabel ;
   Label10 :  TLabel ;
   Label11 :  TLabel ;
   Bevel1 :  TBevel ;
   Database1 :  TDatabase ;
   Table1 :  TTable ;
   DataSource1 :  TDataSource ;
  procedure  FormCreate( Sender:  TObject) ;
```

```
 private
  { Déclarations privées }
  public
  { Déclarations publiques }
  App_Path :string ;
end;

 var
   Form1 :  TForm1 ;

implementation

  {$R *.DFM}

 procedure  TForm1.FormCreate( Sender:  TObject) ;
 begin
    DataBase1.DatabaseName := 'MaBase';
    DataBase1.connected := true ;  //connexion sur la base de données

    Table1.DatabaseName := 'MaBase';
    Table1.TableName := 'TableClients';
    Table1.open ;  //Ouverture de la table attachée au composant Table1
 end;

end.
```

De la même manière que dans l'exercice précédent, le **TDBEdit** et le **TDBNavigator** ont été paramétrés pendant la conception. Ils peuvent aussi être paramétrés par programme pendant l'exécution.

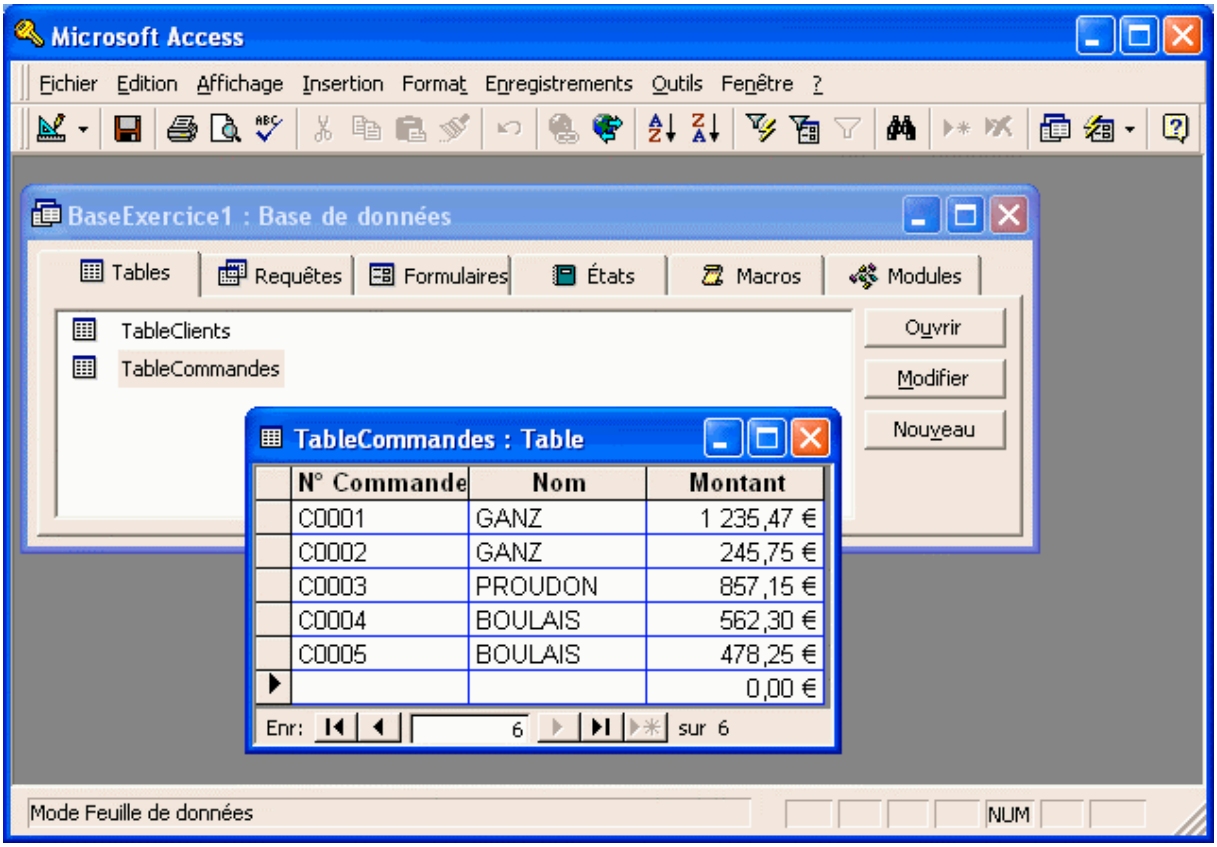
Autre solution de code : le paramétrage des composants lors de la création de la fiche

```
procedure  TForm1.FormCreate( Sender:  TObject) ;
begin
    DataBase1.DatabaseName := 'MaBase';
    DataBase1.connected := true ;  //connexion sur la base de données
    DBNavigator1.DataSource := DataSource1 ;
    DBEdit1.DataSource := DataSource1 ;
    DBEdit2.DataSource := DataSource1 ;
    DBEdit3.DataSource := DataSource1 ;
    DBEdit4.DataSource := DataSource1 ;
    DBEdit5.DataSource := DataSource1 ;
    Table1.DatabaseName := 'MaBase';
    Table1.TableName := 'TableClients';
    Table1.open ;  //Ouverture de la table attachée au composant Table1
end;
```

Ex-4 : Affichage des résultats d'une requête

3°) Requête affichant nom, prénom et montant des achats

Avec Access, dans notre BD nous ajoutons une seconde table : **TableCommandes**, qui contient les achats effectués par les personnes rangées dans la base (considérées comme des acheteurs) :

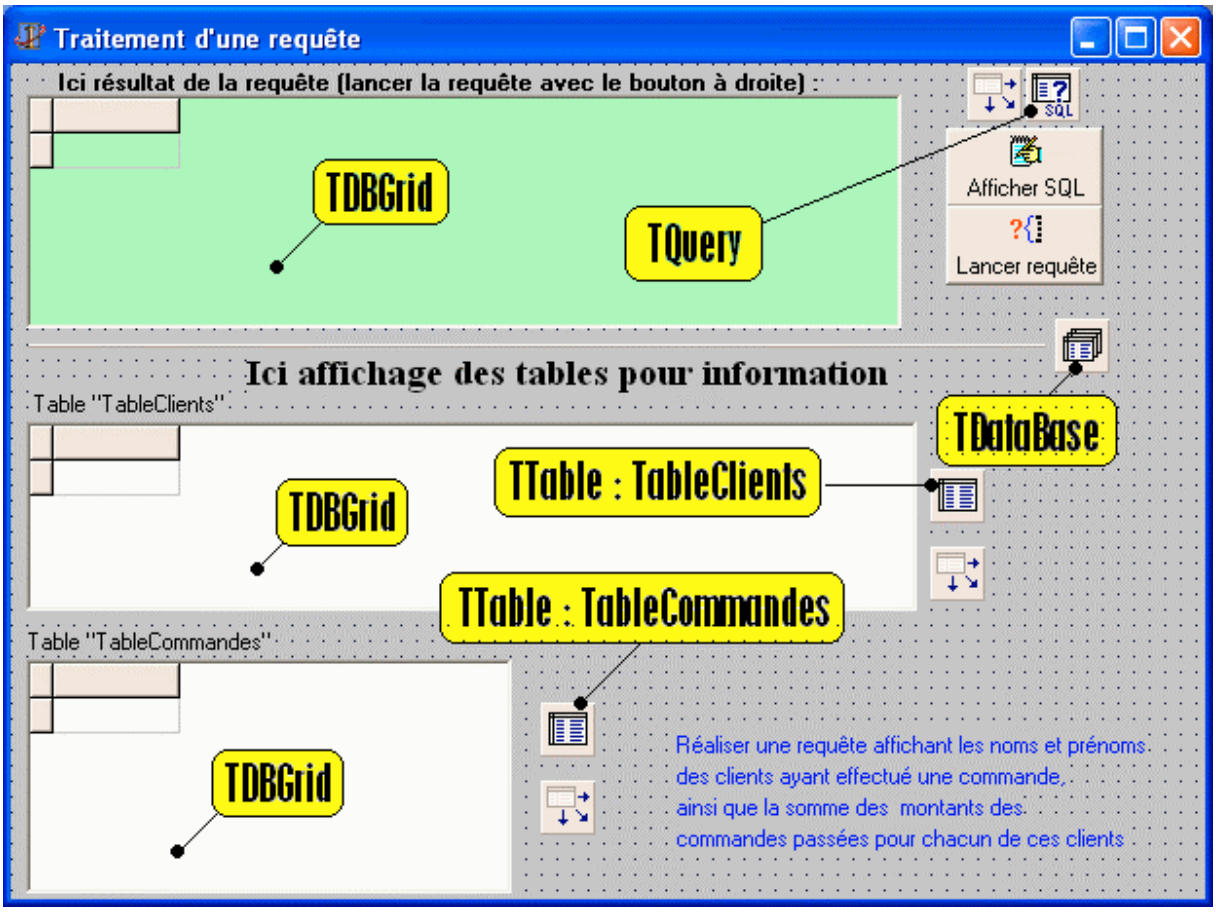


L'interface de l'exercice, dans laquelle aucun des composants déposés n'est paramétré lors de la conception, mais programmé pendant l'exécution afin que le lecteur puisse réutiliser le code source.

4°) Code source delphi de l'exercice

```
unit UBDExo3 ;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBTables, StdCtrls, ExtCtrls, Db, Buttons, Grids, DBGrids ;

type
  TForm1 = class (TForm)
  DBGrid3 : TDBGrid ;
  DBGrid2 : TDBGrid ;
  DBGrid1 : TDBGrid ;
  SpeedButton2 : TSpeedButton ;
  SpeedButton1 : TSpeedButton ;
  Database1 : TDatabase ;
  TableClients : TTable ;
  DSTableClients : TDataSource ;
  TableCommandes : TTable ;
  DSTableCommandes : TDataSource ;
```

```
  Bevel1 : TBevel ;
  DataSource1 : TDataSource ;
  Label4 : TLabel ;
  Label5 : TLabel ;
  Label3 : TLabel ;
  Label1 : TLabel ;
  Label2 : TLabel ;
  Label6 : TLabel ;
  Query1 : TQuery ;
  Label7 : TLabel ;
  Label8 : TLabel ;
```

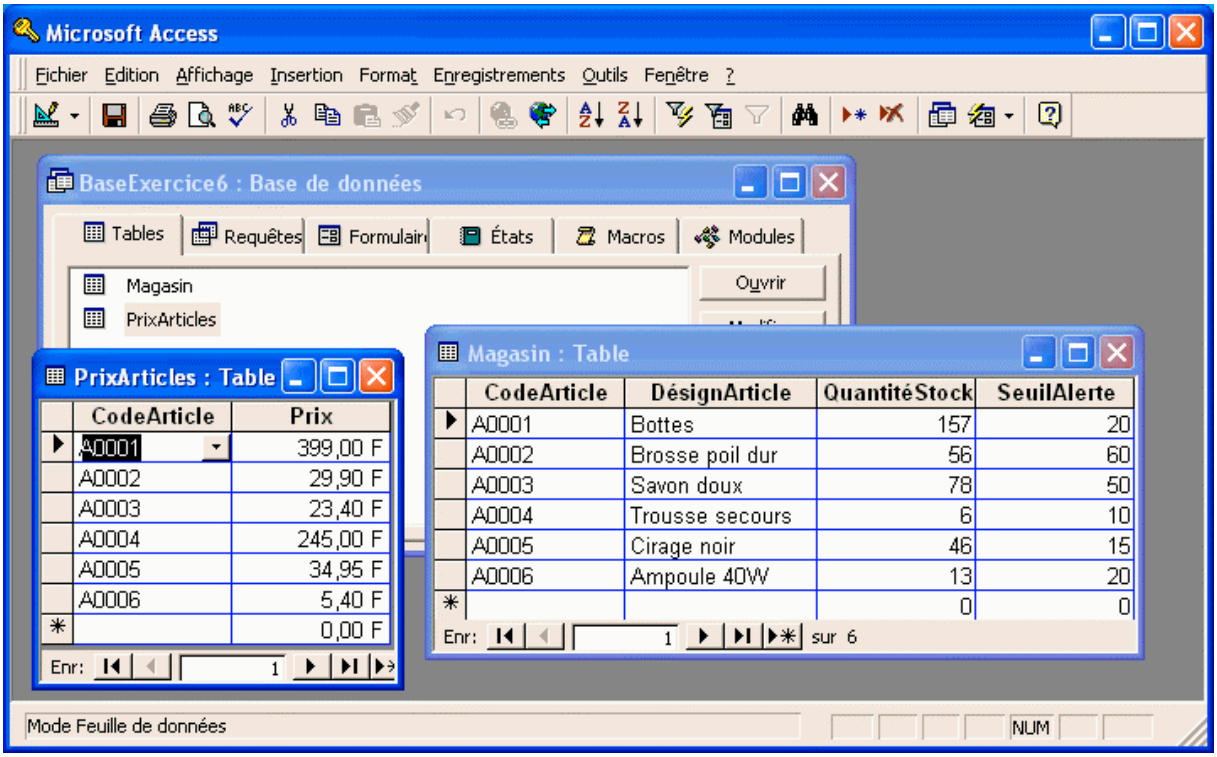



```
  procedure SpeedButton2Click( Sender: TObject) ;
  procedure SpeedButton1Click( Sender: TObject) ;
  procedure FormCreate( Sender: TObject) ;
 private
  { Déclarations privées }
  public
  { Déclarations publiques }
  App_Path :string ;
end;

 var
  Form1 : TForm1 ;

implementation

uses uFAffichage3 ;

  {$R *.DFM}
```

```
 // Affiche dans un Tmemo le contenu du Tstrings Query.SQL (le texte de la requête)
procedure  TForm1.SpeedButton2Click( Sender:  TObject) ;
begin
   FAffichage3.memo1.lines.assign(Query1.SQL) ;
   FAffichage3.showmodal ;
end;

  procedure  TForm1.SpeedButton1Click( Sender:  TObject) ;
  begin
    Query1.open ;  // lance la requête incluse dans le champ SQL du TQuery
   end;

  procedure  TForm1.FormCreate( Sender:  TObject) ;
  begin
    DataBase1.DatabaseName  :=  'MaBase';
    DataBase1.connected  :=  true  ;   //connexion sur la base de données

    TableClients.DatabaseName  :=  'MaBase';
    TableClients.TableName  :=  'TableClients';
    TableClients.open  ;   //Ouverture de la table attachée au composant : TableClients
    DSTableClients.DataSet := TableClients ; // liaison TDBGrid <--> TTable
    DBGrid2.DataSource := DSTableClients ;

    TableCommandes.DatabaseName  :=  'MaBase';
    TableCommandes.TableName  :=  'TableCommandes';
    TableCommandes.open  ;   //Ouverture de la table attachée au composant : TableCommandes
    DSTableCommandes.DataSet := TableCommandes ;  // liaison TDBGrid <--> TTable
    DBGrid1.DataSource := DSTableCommandes ;

    Query1.DatabaseName := 'MaBase';
    Query1.SQL.Clear ;
    Query1.SQL.Append( 'SELECT TableClients.Nom, TableClients.Prénom, ' +
                                  'Sum(TableCommandes.Montant)' ) ;
    Query1.SQL.Append('FROM TableClients INNER JOIN TableCommandes' +
                                 'ON TableClients.Nom = TableCommandes.Nom' ) ;
    Query1.SQL.Append( 'GROUP BY TableClients.Nom, TableClients.Prénom' ) ;
    DataSource1.DataSet := Query1 ;  // liaison TDBGrid <--> TQuery
   end;

end.
```

Ex-5 : Affichage des résultats calculés d'une requête

3°) Requête et affichage de résultats calculés



Si l'on lance la requête SQL demandée :

SELECT Magasin.CodeArticle, Magasin.DésignArticle, Magasin.QuantitéStock, Magasin.SeuilAlerte
FROM Magasin **ORDER BY** Magasin.CodeArticle

vers un TDBGrid comme dans l'exemple précédent, voici ce qui sera affiché :

CodeArticle	DésignArticle	QuantitéStock	SeuilAlerte
A0001	Bottes	157	20
A0002	Brosse poil dur	56	60
A0003	Savon doux	78	50
A0004	Trousse secours	6	10
A0005	Cirage noir	46	15
A0006	Ampoule 40W	13	20

L'affichage de la table TablePrix se fait par liaison avec un TTable et un TDataSource :

CodeArticle	Prix
A0001	399
A0002	29,9
A0003	23,4
A0004	245
A0005	34,95
A0006	5,4

En fait l'énoncé nous demande à partir de la requête précédente de supprimer à l'affichage la colonne "CodeArticle" et d'ajouter deux colonnes en plus, l'une appelée "Message" est

obtenue par calcul sur le niveau du seuil d'alerte, l'autre nommée "Prix" correspond au prix de chaque "Article" contenu dans la table "TablePrix" :

DésignArticle	QuantitéStock	SeuilAlerte	Message	Prix
Bottes	157	20	OK	399,00 €
Brosse poil dur	56	60	ALERTE!!!	29,90 €
Savon doux	78	50	OK	23,40 €
Trousse secours	6	10	ALERTE!!!	245,00 €
Cirage noir	46	15	OK	34,95 €
Ampoule 40W	13	20	ALERTE!!!	5,40 €

Il faut donc créer des données intermédiaires permettant ce dernier affichage sur 5 colonnes.

Au lieu d'envoyer directement dans un TDBGrid le résultat de la requête << **SELECT** Magasin.CodeArticle, Magasin.DésignArticle, Magasin.QuantitéStock, Magasin.SeuilAlerte **FROM** Magasin **ORDER BY** Magasin.CodeArticle >>, nous l'envoyons dans des objets de champ persistant Delphi (cf. doc Delphi).

Un objet de champ persistant est en première approximation un moyen souple de stocker des informations de données, il est équivalent à une sorte de variable locale pour les données de la BD.

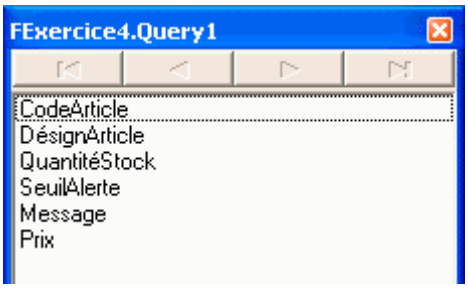
Il nous faut créer 6 objets de champ persistant : 4 pour le résultat de la requête + 1 pour le champ Prix de la TablePrix + 1 nouveau pour le message d'alerte. Chaque champ a un type correspondant exactement au type de données du champ de la BD associée (TstringField pour du texte, TintegerField pour du numérique, TcurrencyField pour du monétaire).

```
Query1CodeArticle: TStringField;
Query1DsignArticle: TStringField;
Query1QuantitStock: TIntegerField;
Query1SeuilAlerte: TIntegerField;
Query1Message: TStringField;
Query1Prix: TCurrencyField;
```

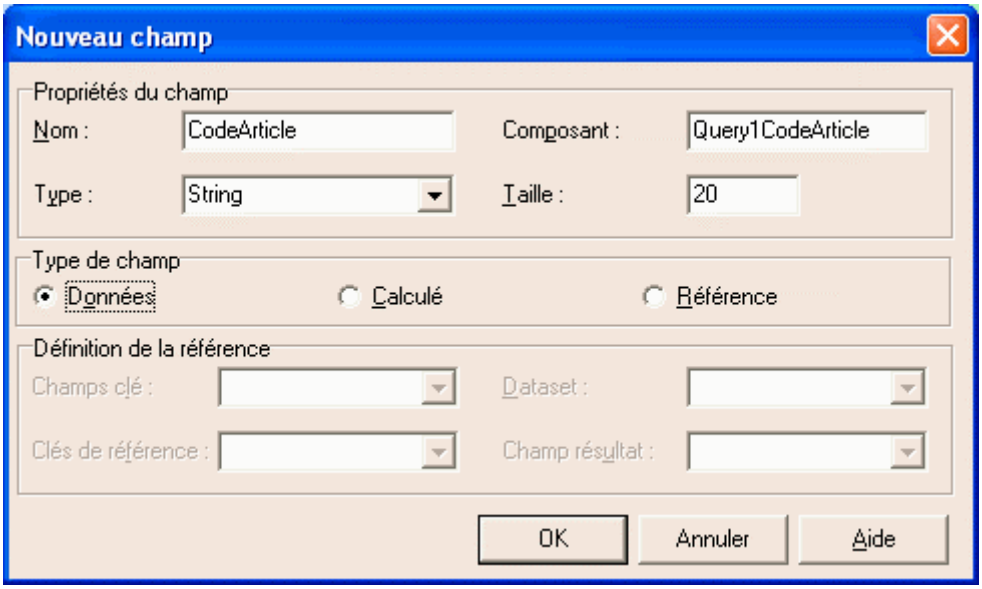
La documentation Delphi indique comment créer par programme de tels champ, par exemple ci-dessous la création de l'objet Query1CodeArticle: TStringField; lié au Tquery : Query1 :

```
Query1.Close;
Query1DsignArticle:= TStringField.Create(Self); // self représente la Tform de dépôt du Query1
Query1DsignArticle.FieldName := 'DesignArticle';
Query1DsignArticle.Index := Query1.FieldCount;
Query1DsignArticle.DataSet := Query1;
 Query1.FieldDefs.UpDate;
 Query1.Open;
```

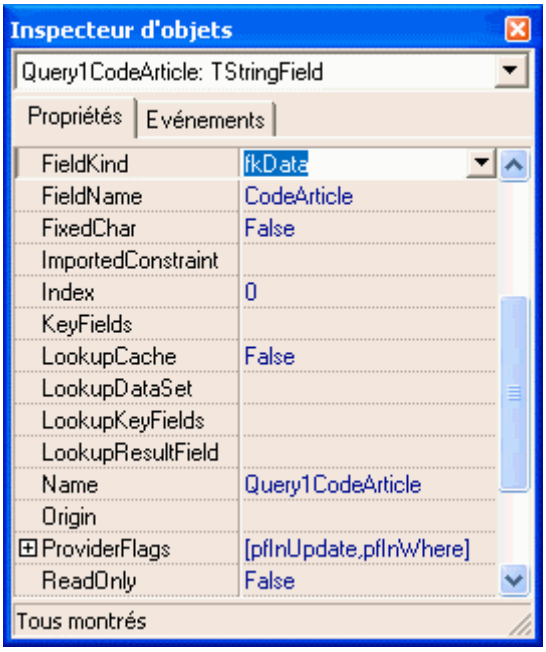
Il en est de même pour les 5 autres objets de champ. Delphi permet une création directement pendant la conception par double click sur le composant Query1, une interface de saisie d'une liste d'objets de champ apparaît. Ci dessous la liste obtenue après saisie des 6 objets :



Le premier objet de champ est saisi comme suit :

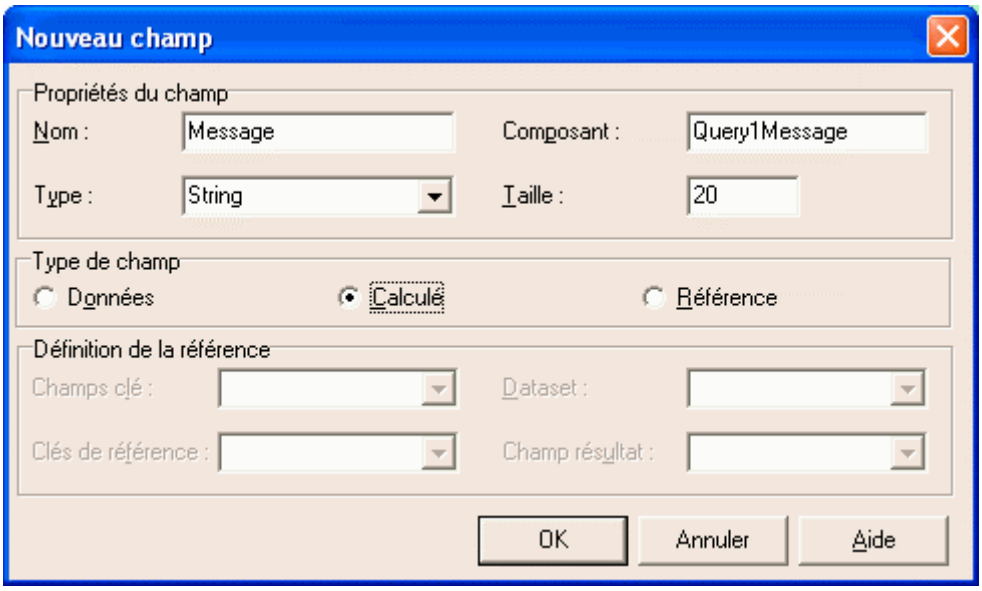


Voici sa vue dans l'inspecteur d'objet :

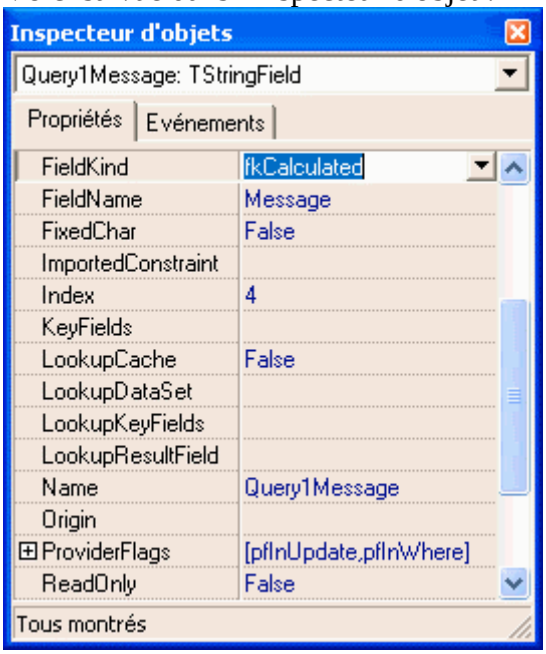


Les trois objets de champ suivants sont sur le même modèle (type de champ : données).

Le cinquième champ de Message est calculé :



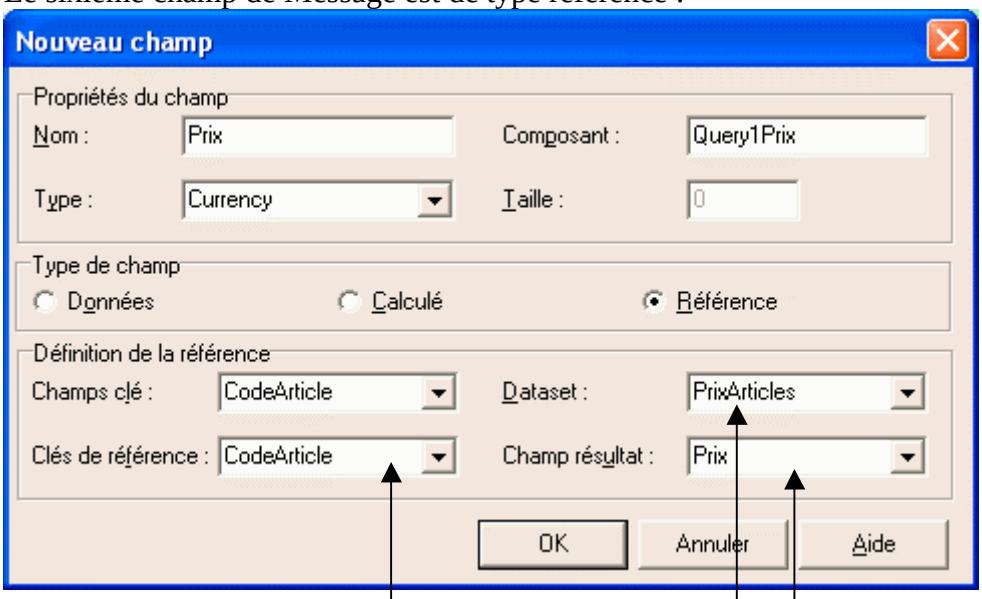
Voici sa vue dans l'inspecteur d'objet :



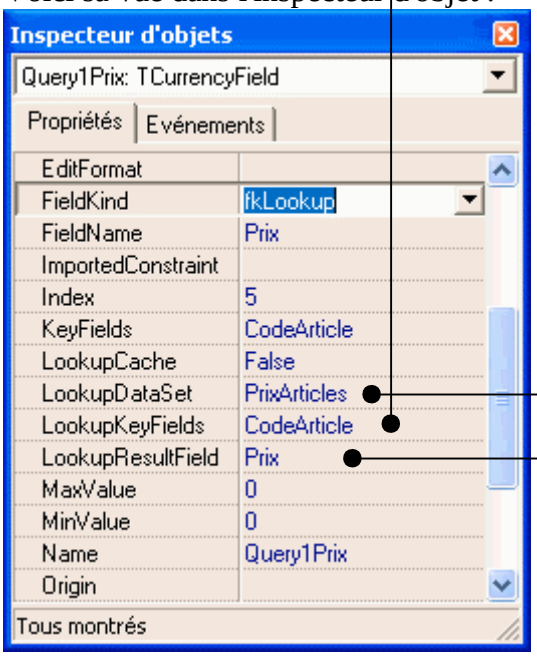
Le sixième et dernier champ de Prix est une référence à un champ existant dans la TablePrix, il est donc nécessaire de fournir :

- le nom de la table,
- le champ de clef primaire,
- le nom du champ où l'on va chercher la valeur correspondant à la clef

Le sixième champ de Message est de type référence :



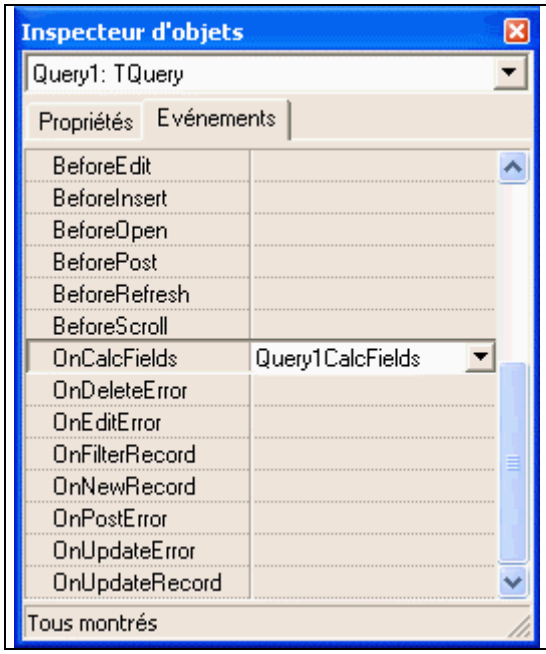
Voici sa vue dans l'inspecteur d'objet :



Voici le code de calcul d'alerte, permettant de mettre la valeur du message d'alerte en fonction du seuil d'alerte prévu :

```
if  Query1QuantitStock.value <= Query1SeuilAlerte.value  then
    Query1Message.value := 'ALERTE!!!'
else
    Query1Message.value := 'OK'
```

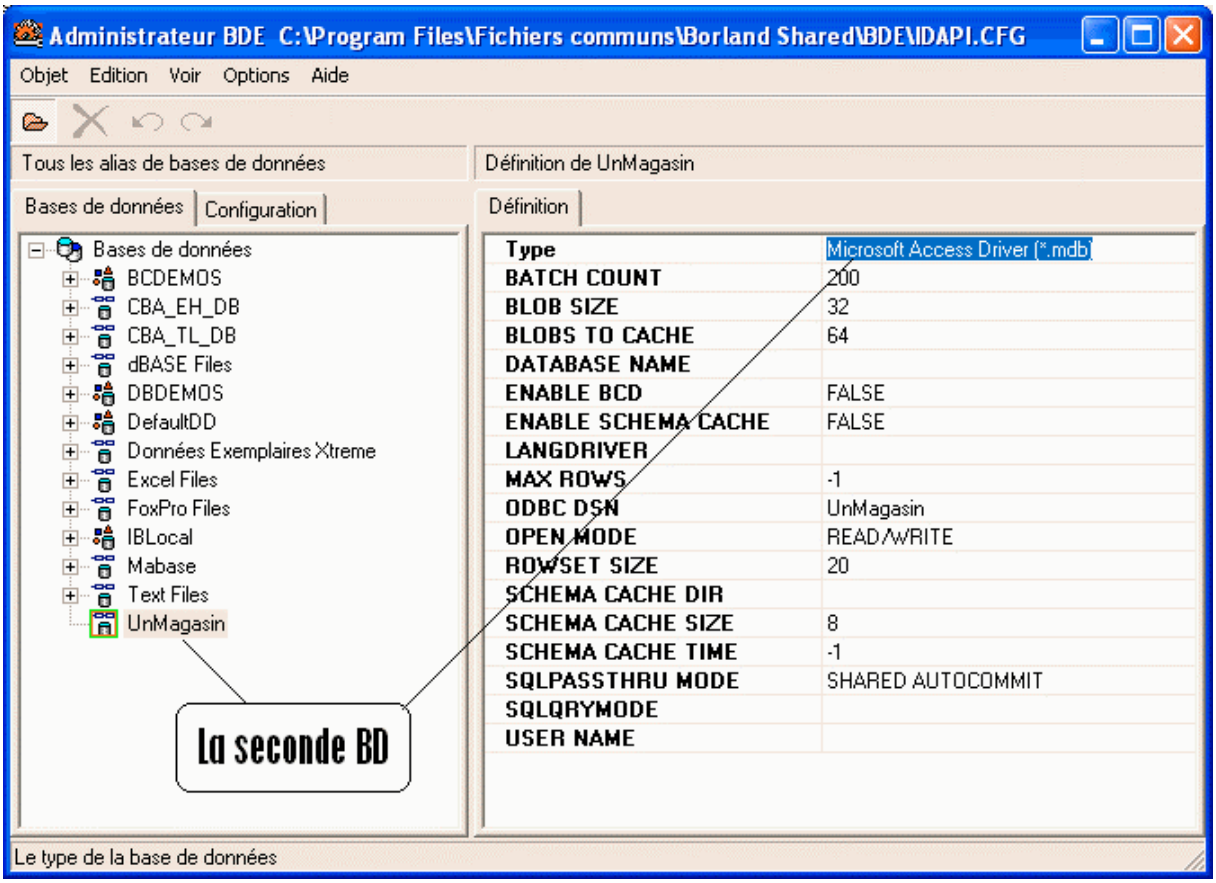
Le composant Query1 de type TQuery possède une événement OnCalcFields qui se produit lorsque l' application évalue les champs calculés. Le champ Query1Message est un champ calculé donc le code de calcul d'alerte peu être opportunément exécuté à chaque fois que la requête est lancée :



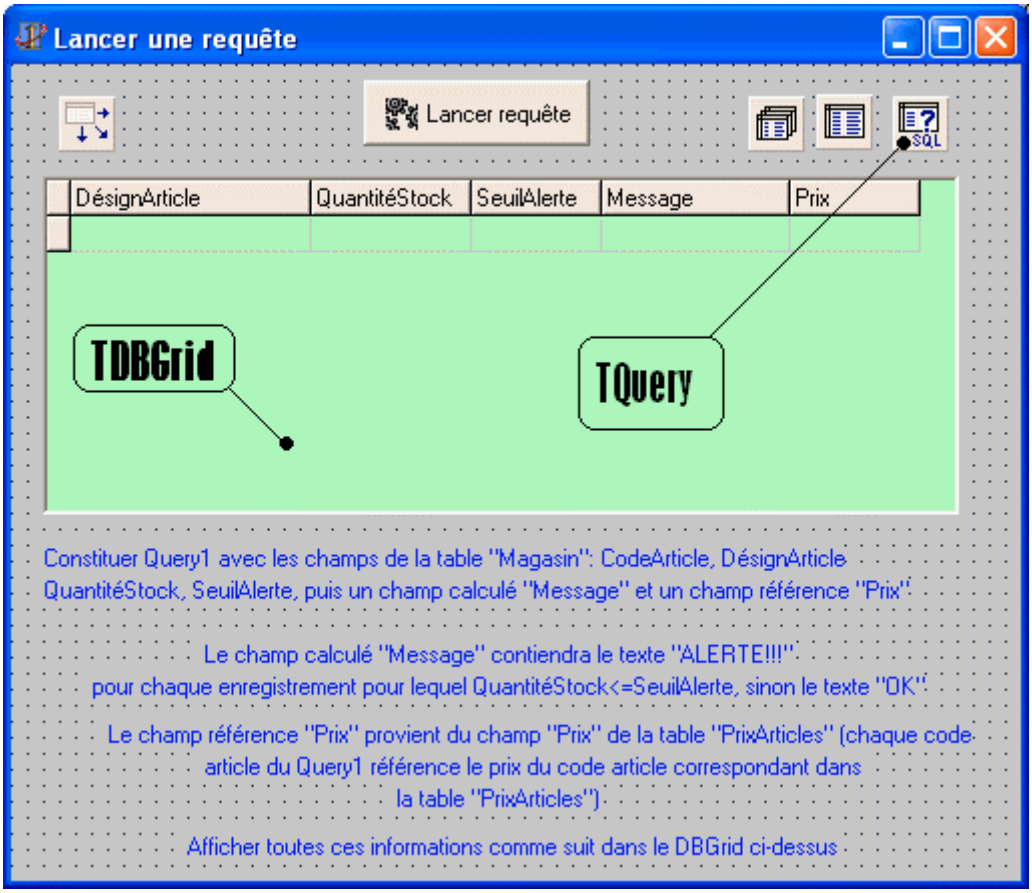

```
procedure Tform1.Query1CalcFields (DataSet: TDataSet);
begin
    // le code de calcul d'alerte
end;
```

4°) Code source delphi de l'exercice

Nous avons créé un nouvel alias de connexion physique par pilote ODBC dénommé "UnMagasin", pour une autre BD associée au fichier physique « **BaseExercice6.mdb** » :

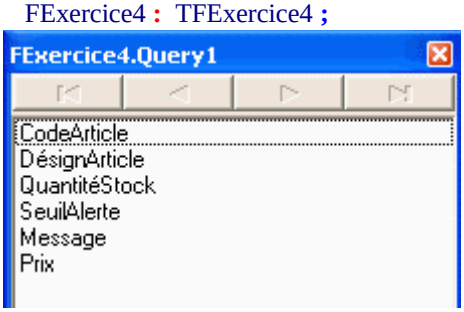


```
unit UBDExo4 ;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
 Buttons, Grids, DBGrids, Db, DBTables, StdCtrls ;

type
  TFExercice4 = class (TForm)
 Database1 : TDatabase ;
  Query1 : TQuery ;
  DataSource1 : TDataSource ;
  DBGrid1 : TDBGrid ;
  SpeedButton1 : TSpeedButton ;
  Query1DsignArticle : TStringField ;
  Query1QuantitStock : TIntegerField ;
  Query1SeuilAlerte : TIntegerField ;
  Query1Message : TStringField ;
  Label1 : TLabel ;
  Label2 : TLabel ;
  Label3 : TLabel ;
  Label4 : TLabel ;
  Label5 : TLabel ;
  Label6 : TLabel ;
  Label7 : TLabel ;
  Query1CodeArticle : TStringField ;
  PrixArticles : TTable ;
  Query1Prix : TCurrencyField ;
  Label8 : TLabel ;
```




```
procedure SpeedButton1Click( Sender: TObject) ;
```

```
  procedure  FormCreate( Sender:  TObject) ;
  procedure  Query1CalcFields(DataSet :  TDataSet) ;
 private
  { Déclarations privées }
  public
  { Déclarations publiques }
end;

 var
   FExercice4 :  TFExercice4 ;
```




```
{

 On a créé comme indiqué plus haut, les 6 champs
 d'objets persistants pendant la conception .

}


implementation
var
  rep_appli :string ;

  {$R *.DFM}

 procedure  TFExercice4.SpeedButton1Click( Sender:  TObject) ;
 begin
   Query1.Open ;
 end;

 procedure  TFExercice4.FormCreate( Sender:  TObject) ;
 begin
    DataBase1.DatabaseName   :=   'UnMagasin';
    DataBase1.connected   :=   true   ;    //connexion sur la base de données

    PrixArticles.DatabaseName   :=   'UnMagasin';  //TTable connecté à la BD
    PrixArticles.TableName := 'PrixArticles';  // TTable connecté à  la table PrixArticles de la BD

    DBGrid1.DataSource := DataSource1 ;  // liaison TDBGrid <--> TDataSource

    Query1.DatabaseName  :=  'UnMagasin';
    Query1.SQL.Clear  ;
    Query1.SQL.Append(  'SELECT Magasin.CodeArticle, Magasin.DésignArticle,'  +
                                        'Magasin.QuantitéStock, Magasin.SeuilAlerte'  )  ;
    Query1.SQL.Append( 'FROM Magasin ORDER BY Magasin.CodeArticle' )  ;
    DataSource1.DataSet  :=  Query1  ;   // liaison TDataSource <--> TQuery
  end;

 procedure  TFExercice4.Query1CalcFields(DataSet :  TDataSet) ;
 begin
  if  Query1QuantitStock.value <= Query1SeuilAlerte.value  then
    Query1Message.value := 'ALERTE!!!'
  else
    Query1Message.value := 'OK'
 end;
end.
```

Etape n°1 le processus d'analyse

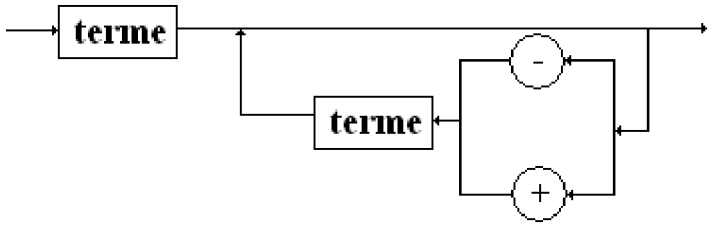
Nous suivons la stratégie d'analyse évoquée au chapitre 7.2.

- **Afin d'améliorer le traitement des données, nous ajoutons une sentinelle à l'expression, soit le caractère '.' choisi comme sentinelle.**
- **Afin de simplifier le code (l'objectif principal étant la construction de l'arbre abstrait) nous n'analysons que des expressions correctes, donc nous ne nous préoccuperons pas des tests d'erreurs.**

Nous procédons donc règles par règles et nous donnons au lecteur l'algorithme et l'implantation en Delphi de chaque bloc associé à une règle.

étude de la Règle-1 :

<expr> :



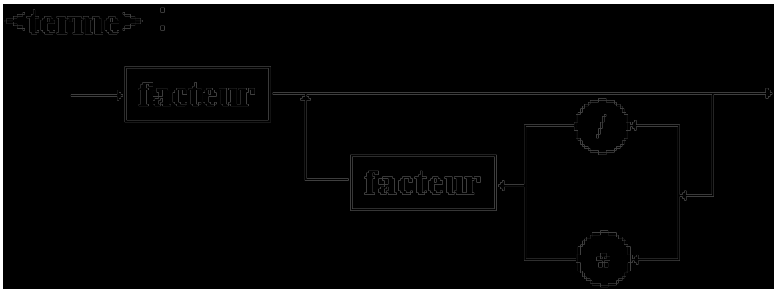
Bloc Analyser expr :

```
si SymLu ∈ Init(terme) alors
 Analyser terme ;
 tantque SymLu ∈ Init( {+,-} )faire
   Symsuivant;
   Analyser terme ;
 ftant;
 //si SymLu ∉ Init( {'.'} )Alors Erreur fsi
sinon Erreur
fsi
```

Implantation en Delphi :

```
procedure expr;
begin
   terme;
   while SymLu in ['+', '-'] do
     begin
       Symsuivant;
       terme;
     end
end;{expr}
```

étude de la Règle-2 :



Bloc Analyser terme :

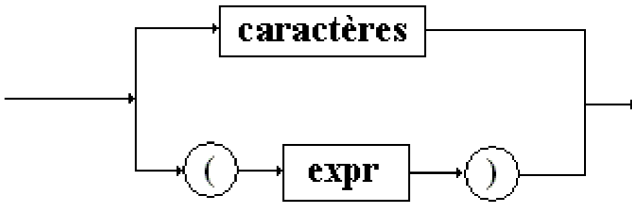
```
si SymLu ∈ Init(facteur) alors
 Analyser facteur ;
 tantque SymLu ∈ Init( { *, / } )faire
   Symsuivant;
   Analyser facteur ;
 ftant;
 //si SymLu ∉ Init( {+,-,*,/,'.'} )Alors Erreur fsi
fsi
```

Implantation en Delphi :

```
procedure terme;
begin
     facteur;
     while SymLu in ['*', '/'] do
       begin
         Symsuivant;
         facteur;
       end
 end;{terme}
```

étude de la Règle-3 :

<facteur> :



Bloc Analyser facteur :

```
si SymLu ∈ Init(caractères) alors
 Symsuivant
sinon
 si SymLu ∈ Init({'('}) alors
   Symsuivant;
   Analyser expr ;
   si SymLu ∈ Init({')'}) alors
     Symsuivant;
   fsi
 sinon Erreur
 fsi
fsi
```

Implantation en Delphi :

```
procedure facteur ;
begin
    if SymLu = '(' then
      begin
        Symsuivant;
        expr;
        if SymLu = ')' then
          Symsuivant
      end
    else // on est dans caractères car: expression correcte
      begin
        Symsuivant
      end
end;{facteur}
```

programme complet en Delphi :

Voici le programme Delphi obtenu pour analyser une expression correctement écrite, pour l'instant si l'on exécute ce programme tel quel, il ne produira rien, car nous sommes encore à la première étape du travail.

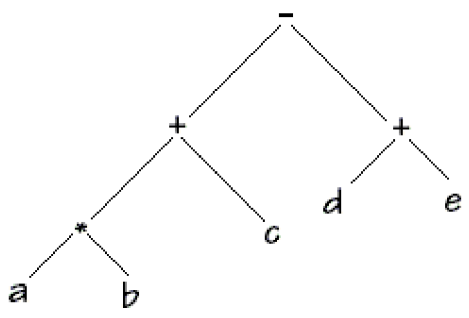
```
program expression;
var
    SymLu: char;
    numcar: integer;
    chaine: string;

  procedure init;
  begin
    numcar := 0;
  end;

  procedure Symsuivant;
  begin
    numcar := numcar + 1;
    SymLu := chaine[numcar]
  end;

procedure expr;
begin
    terme;
    while SymLu in ['+', '-'] do
      begin
        Symsuivant;
        terme;
      end
end;{expr}

procedure terme;
begin
      facteur;
      while SymLu in ['*', '/'] do
        begin
          Symsuivant;
          facteur;
        end
  end;{terme}
```

```
procedure facteur ;
begin
    if SymLu = '(' then
      begin
        Symsuivant;
        expr;
        if SymLu = ')' then
          Symsuivant
      end
    else // on est dans caractères car: expression correcte
      begin
        Symsuivant
      end
end;{facteur}

begin
  init;
  writeln('entrez une expression:');
  readln(chaine);
  chaine := concat(chaine, '.');
  Symsuivant;
  expr(x);
end.
```

Etape n°2 la construction de l'arbre abstrait

Nous allons stocker les différents symboles de l'expression au fur et à mesure de son analyse, dans un arbre binaire représentant l'arbre abstrait de l'expression comme nous l'avons déjà vu au chapitre 4.2 .

Pour l'exemple à partir de l'expression a*b + c-(d+e) nous construirons l'arbre abstrait figuré ci-dessous :



Implantation de la structure en Delphi avec variables dynamiques :

```
type
  parbre = ^arbre;
  arbre = record
            val: char;
            g, d: parbre
      end;
```

Nous proposons d'écrire une fonction de construction d'un arbre (un constructeur d'arbre)

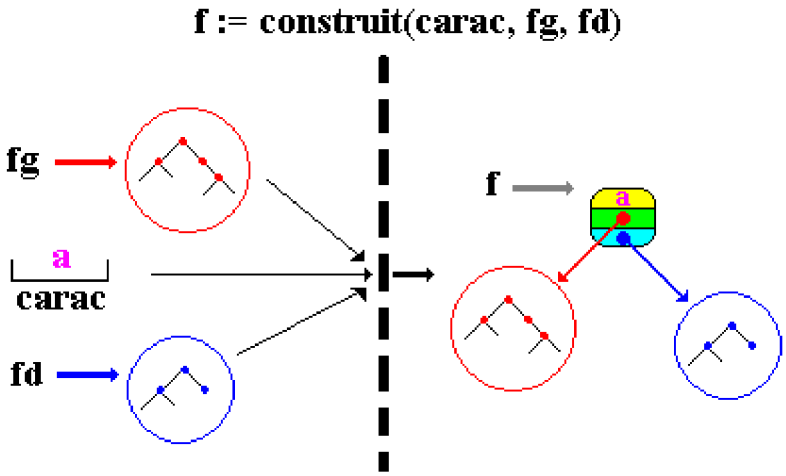
- La fonction reçoit en entrée 3 paramètres correspondant au contenu d'un noeud :
  - la valeur (un char ici),
  - le pointeur (ou référence) vers le fils gauche de ce noeud,
  - et le pointeur (ou référence) vers le fils droit de ce noeud.
- La fonction renvoie un fois le noeud construit, un pointeur (ou référence) sur le noeud nouvellement construit.

Voici le code de la fonction 'Construit' proposée selon le type parbre défini plus haut :

```
function Construit (c: char; fg, fd: parbre): parbre;
var  floc: parbre;
begin
   new(floc);
   with floc^ do
   begin
      val := c;
      g := fg;
      d := fd
   end;
   construit := floc
end;{construit}
```

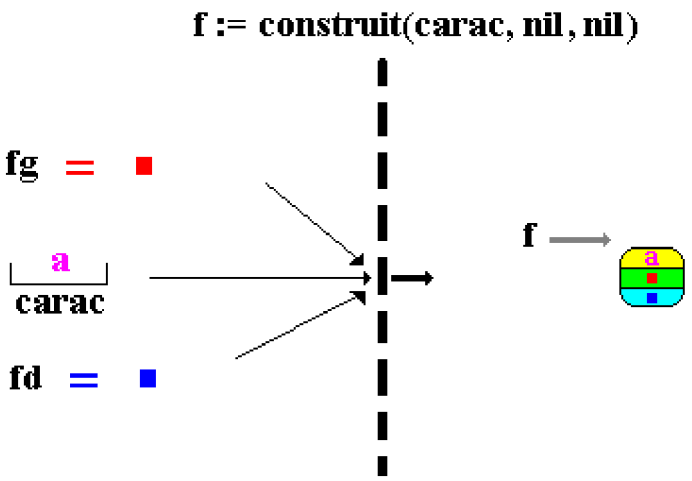
Ci-dessous sur deux exemples montrons ce que produit la **function** Construit(..).

Exemple -1 : la construction d'un noeud ayant deux descendants
(un sous-arbre gauche nommé **fg**, et un sous-arbre droit nommé **fd**)



A la fin de la construction la fonction construit a relié l'arbre gauche rouge avec l'arbre droit bleu dans le noeud pointé par **f**, respectivement comme sous-arbre gauche rouge et comme sous-arbre droit bleu, le champ valeur du noeud **f** contient le caractère '**a**'.

Exemple -2 : la construction d'une feuille



génération pour la Règle-1 :

Bloc générer arbre - expr :

```
si SymLu ∈ Init(terme) alors
 Analyser terme ;
 tantque SymLu ∈ Init( {+,-} )faire
   Symsuivant;
   Analyser terme ;
   construire arbre
 ftant;
 //si SymLu ∉ Init( {'.'} )Alors Erreur fsi
sinon Erreur
fsi
```

Implantation en Delphi :

```
procedure expr (var f: parbre) ;
var
      carloc: char;
      ft: parbre;
 begin
    terme(f);
    while SymLu in ['+', '-'] do
      begin
        carloc := SymLu;
        Symsuivant;
        terme(ft);
        f := construit(carloc, f, ft){construction de l'arbre}
      end
end;{expr}
```

génération pour la Règle-2 :

Bloc générer arbre - terme :

```
si SymLu ∈ Init(facteur) alors
 Analyser facteur ;
 tantque SymLu ∈ Init( { *, / } )faire
   Symsuivant;
   Analyser facteur ;
```

```
  construire arbre
 ftant;
 //si SymLu ∉ Init( {+,-,*,/,'.'} )Alors Erreur fsi
fsi
```

Implantation en Delphi :

```
procedure terme (var f: parbre);
    var
      carloc: char;
      floc: parbre;
   begin
     facteur(f);
     while SymLu in ['*', '/'] do
       begin
         carloc := SymLu;
         Symsuivant;
         facteur(floc);
         f := construit(carloc, f, floc){construction de l'arbre}
       end
   end;{terme}
```

génération pour la Règle-3 :

Bloc générer arbre - facteur :

```
si SymLu ∈ Init(caractères) alors
 Symsuivant
sinon
 si SymLu ∈ Init({'('}) alors
   Symsuivant;
   Analyser expr ;
   si SymLu ∈ Init({')'}) alors
     Symsuivant;
   fsi
 sinon Erreur
 fsi
fsi
```

Implantation en Delphi :

```
procedure facteur (var f: parbre);
begin
    if SymLu = '(' then
      begin
        Symsuivant;
        expr(f);
        if SymLu = ')' then
          Symsuivant
      end
    else
      begin
        f := construit(SymLu, nil, nil); {construction de l'arbre }
        Symsuivant
      end
end;{facteur}
```

programme complet en Delphi :

Voici un programme Delphi obtenu pour engendrer un arbre abstrait d'une expression correctement écrite; ici les procédures terme et facteur ont été incluses dans la partie déclaration de la procédure expr, mais elles peuvent être dissociées et se retrouver au même niveau de déclaration sans rien changer au fonctionnement du programme.

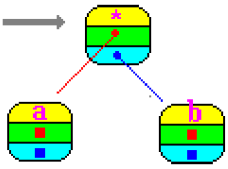
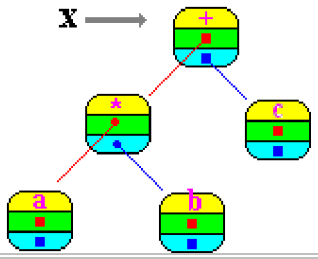
```
program expression;
{les expressions entrées sont correctes }
  type
    parbre = ^arbre;
    arbre = record
                val: char;
                g, d: parbre
          end;
  var
    x: parbre;
    SymLu: char;
    numcar: integer;
    chaine: string;

  procedure init;
  begin
    numcar := 0;
    x := nil;
  end;

  function construit (c: char; fg, fd: parbre): parbre;
    var
      floc: parbre;
  begin
    new(floc);
    with floc^ do
      begin
        val := c;
        g := fg;
        d := fd
      end;
    construit := floc
  end;{construit}

  procedure Symsuivant;
  begin
    numcar := numcar + 1;
    SymLu := chaine[numcar]
  end;

 {construction d'un arbre d'expressions à partir de la grammaire :}
  procedure expr (var f: parbre);
    var
      carloc: char;
      ft: parbre;
    procedure facteur (var f: parbre);
    begin
      if SymLu = '(' then
        begin
          Symsuivant;
          expr(f);
          if SymLu = ')' then
            Symsuivant
        end
      else
```

```
      begin
        f := construit(SymLu, nil, nil); {construction de l'arbre }
        Symsuivant
      end
    end;{facteur}

    procedure terme (var f: parbre);
      var
        carloc: char;
        floc: parbre;
    begin
      facteur(f);
      while SymLu in ['*', '/'] do
        begin
          carloc := SymLu;
          Symsuivant;
          facteur(floc);
          f := construit(carloc, f, floc){construction de l'arbre}
        end
    end;{terme}

  begin{expr}
    terme(f);
    while SymLu in ['+', '-'] do
      begin
        carloc := SymLu;
        Symsuivant;
        terme(ft);
        f := construit(carloc, f, ft){construction de l'arbre}
      end
  end;{expr}

begin
  init;
  writeln('entrez une expression:');
  readln(chaine);
  chaine := concat(chaine, '.');
  Symsuivant;
  expr(x);
end.
```

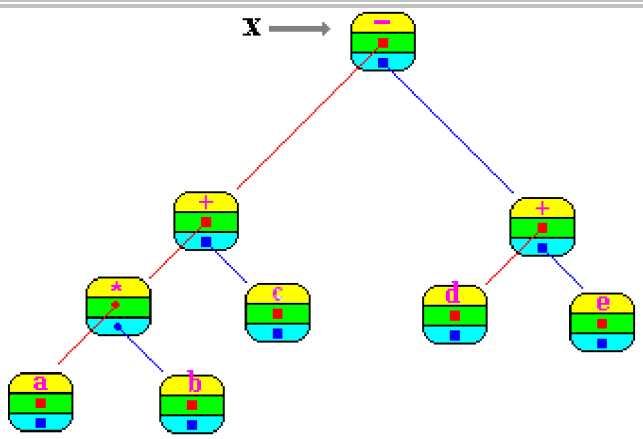
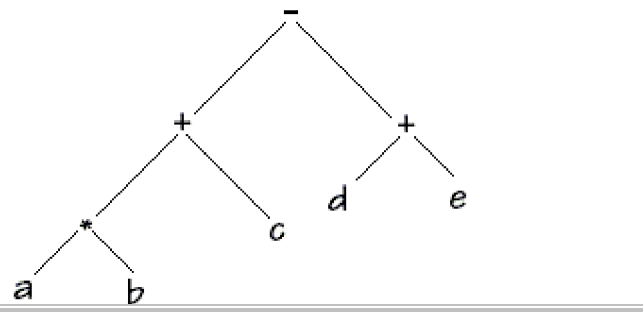
Suivons à titre d'exemple la construction par le programme précédent, de l'arbre abstrait de l'expression déjà citée plus haut soit : **a*b + c-(d+e)**.

Nous effectuons une trace pas à pas du début de l'exécution sur l'expression :

Procédures appelées	Résultat produit
readln(chaine);	chaine = **a*b + c-(d+e)**.
init;	x = **nil**
Symsuivant	SymLu = **a** , dans **a*b + c-(d+e)**.
expr(x) |terme(x) | |facteur(x) | | |x := construit(SymLu, **nil**, **nil**); | | |Symsuivant	**x** → SymLu = * , dans **a*b + c-(d+e)**.
expr(x)	carloc = *

|terme(x) | |facteur(x) | |SymLu **in** ['*', '/'] = true | carloc := SymLu; | Symsuivant; | facteur(floc);	SymLu = **b** , dans **a*b + c-(d+e)**. floc = **nil**
expr(x) |terme(x) | |facteur(floc); | | |floc := construit(SymLu, **nil**, **nil**); | | |Symsuivant	*carloc* = * **floc** → SymLu = + , dans **a*b** + **c-(d+e)**.
expr(x) |terme(x) | |facteur(floc); | |x := construit(carloc, x, floc)	*carloc* = * **x** → 
expr(x) |terme(x); |SymLu **in** ['+', '-'] = true | carloc := SymLu; | Symsuivant; | terme(ft);	carloc = + SymLu = **c** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x) | |terme(ft) | | |facteur(ft); | | | |ft := construit(SymLu, **nil**, **nil**); | | | |Symsuivant	*carloc* = + **ft** → SymLu = - , dans **a*b** + **c - (d+e)**.
expr(x) |terme(x) | |terme(ft) | | |facteur(ft); | | |x := construit(carloc, x, ft)	*carloc* = + **x** → 
expr(x) |terme(x) | |terme(ft) | |facteur(ft) | | |SymLu **in** ['*', '/'] = false |SymLu **in** ['+', '-'] = true | carloc := SymLu; | Symsuivant; | |terme(ft);	carloc = - SymLu = **(** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x) | |facteur(ft); | | |SymLu = '(' = true | | Symsuivant; | | |expr(ft); | | |terme(ft); | | | |facteur(ft);	carloc = - SymLu = **d** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x)	*carloc* = -

| |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |facteur(ft) | | | | | |ft := construit(SymLu, **nil**, **nil**); | | | | | |Symsuivant;	**ft** → d SymLu = **+** , dans **a*b + c-(d+e)**.
expr(x) |terme(x) | |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |facteur(ft) | | | | |SymLu **in** ['*', '/'] = false | | | |SymLu **in** ['+', '-'] = true | | | | carloc := SymLu; | | | | Symsuivant; | | | | |terme(ft); | | | | | |facteur(ft);	*dans la pile :* ---------- *carloc* = - **ft** → d ---------- carloc = + SymLu = **e** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x) | |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |terme(ft) | | | | | |facteur(ft) | | | | | | |ft := construit(SymLu, **nil**, **nil**); | | | | | | |Symsuivant;	*dans la pile :* ---------- *carloc* = - **ft** → d ---------- carloc = + **ft** → e SymLu = **)** , dans **a*b + c-(d+e)**.
expr(x) |terme(x) | |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |terme(ft) | | | | | |facteur(ft) | | | | | | |ft := construit(carloc , *ft* , ft) | | | | | |SymLu **in** ['*', '/'] = false	*dans la pile :* ---------- *carloc* = - **x** → + ; * ; c ; a ; b ---------- **ft** → + ; d ; e
expr(x) |terme(x) | |facteur(ft)	

| | |expr(ft) | | | |terme(ft) | | | |SymLu **in** ['+', '-'] = false | |x := construit(carloc, x, ft) |SymLu **in** ['*', '/'] = false Fin	
Arbre abstrait obtenu après exécution :	

Si le lecteur souhaite visualiser les contenus des arbres abstraits, il peut utiliser les 3 parcours en profondeur vus au chapitre 4.3, nous listons ci-dessous les 3 procédures associées au type parbre de l'exercice ainsi que le corps du programme principal appellant ces procédures :

```
program expression;
{les expressions sont correctes et elles sont lues sous les 3 formes
in,post et pré-fixées }
  type
    parbre = ^arbre;
    arbre = record
                val: char;
                g, d: parbre
          end;
  var
    x: parbre;
    SymLu: char;
    numcar: integer;
    chaine: string;

  procedure init;
  begin
    numcar := 0;
    x := nil;
  end;

  function construit (c: char; fg, fd: parbre): parbre;
    var
      floc: parbre;
  begin
    new(floc);
    with floc^ do
      begin
```

```
        val := c;
        g := fg;
        d := fd
      end;
    construit := floc
  end;{construit}

  procedure Symsuivant;
  begin
    numcar := numcar + 1;
    SymLu := chaine[numcar]
  end;

 {construction d'un arbre d'expressions à partir de la grammaire :}
  procedure expr (var f: parbre);
    var
      carloc: char;
      ft: parbre;
    procedure facteur (var f: parbre);
    begin
      if SymLu = '(' then
        begin
          Symsuivant;
          expr(f);
          if SymLu = ')' then
            Symsuivant
        end
      else
        begin
          f := construit(SymLu, nil, nil); {construction de l'arbre }
          Symsuivant
        end
    end;{facteur}

    procedure terme (var f: parbre);
      var
        carloc: char;
        floc: parbre;
    begin
      facteur(f);
      while SymLu in ['*', '/'] do
        begin
          carloc := SymLu;
          Symsuivant;
          facteur(floc);
          f := construit(carloc, f, floc){construction de l'arbre}
        end
    end;{terme}

  begin{expr}
    terme(f);
    while SymLu in ['+', '-'] do
      begin
        carloc := SymLu;
        Symsuivant;
        terme(ft);
        f := construit(carloc, f, ft){construction de l'arbre}
      end
  end;{expr}

 procedure edite (f: parbre);
```

```
  {infixe parenthesee}
  begin
    if f <> nil then
      with f^ do
        begin
          write('(');
          edite(g);
          write(val);
          edite(d);
          write(')')
        end
  end;

  procedure postfixe (f: parbre);
 {sert dans les machine a piles a la compilation}
  begin
    if f <> nil then
      with f^ do
        begin
          postfixe(g);
          postfixe(d);
          write(val);
        end
  end;

  procedure prefixe (f: parbre);
  begin
    if f <> nil then
      with f^ do
        begin
          write(val);
          prefixe(g);
          prefixe(d)
        end
  end;

  procedure infixe (f: parbre);
  begin
    if f <> nil then
      with f^ do
        begin
          infixe(g);
          write(val);
          infixe(d);
        end
  end;

 begin
  init;
  writeln('entrez une expression:');
  readln(chaine);
  chaine := concat(chaine, '.');
  Symsuivant;
  expr(x);
  writeln('Expression parenthésée:');
  edite(x);
  writeln;
  writeln('Expression notation infixée:');
  infixe(x);
  writeln;
```

```
    writeln('Expression notation postfixée:');
    postfixe(x);
    writeln;
    writeln('Expression notation préfixée:');
    prefixe(x);
    writeln
  end.
```

Chapitre 8 : les composants sont des logiciels réutilisables

8.1 Construction de composant avec Delphi

- Déivation à partir d'un composant visuel
- Construction par association de composants visuels
- Construction d'un composant non-visuel

8.2 Les messages Windows avec Delphi

- La programmation dirigée par les messages
- Mécanisme de la répartition des messages
- Création et envoi de messages

8.3 Création d'un événement associé à un message

- Rappel sur la construction d'un événement
- Exemple de création d'événements dans un TEdit

8.4 ActiveX avec la technologie COM

- Les notions d'interfaces COM de microsoft
- ActiveX est un objet COM
- Création d'un ActiveX avec Delphi
- Déploiement et utilisation Web d'une fiche ActiveX

Chapitre 8.1 Construction de composant avec Delphi

Plan du chapitre:

Introduction

1. Dérivation à partir d'un composant visuel

1.1 Ajout de méthodes à un composant d'arbre
1.2 Ajout de propriétés au composant d'arbre

2. Construire par association de composants visuels

2.1 Le composant de visualisation d'arbre TWinArbre
2.2 Evénement OnChange de TWinArbre
2.3 Evénement OnMouseDown de TWinArbre
2.4 Le composant final TWinArbre

2.4.1 Le composant TWinArbre peut se redimensionner

2.5 Le composant de saisie d'expression arithmétique

3. Construire un composant non visuel

3.1 Le composant TListe
3.2 Utilisation du composant TListe

Introduction

Ce chapitre est consacré à la production de composants logiciels réutilisables. Nous allons construire en Delphi trois genres d'exemples de " kits de logiciels " réutilisables et donc créer trois composants que vous pourrez améliorer ou modifier. Une fois terminé, chaque composant sera placé dans la palette des outils composants de Delphi ou Kylix (version Linux de Delphi).

Nous proposons encore une fois une démarche méthodique afin de construire certains de ces " kits ". Pour des composants visuels, nous nous limitons à la construction de composants par dérivation de composants existants. Nous montrons comment ajouter des propriétés ou des méthodes à un composant existant. Nous montrons aussi comment associer plusieurs composants visuels de Delphi pour construire un nouveau composant visuel. Nous construisons à la fin un composant non visuel.

1. Dérivation à partir d'un composant visuel

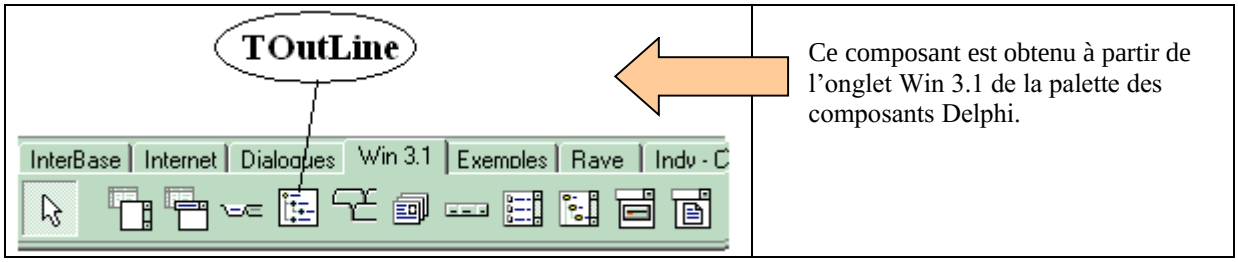
Nous allons construire un nouveau composant qui héritera d'une classe déjà existante de la VCL (Visual Composant Library), nous prendrons un composant visuel d'arbre. Notre action consistera essentiellement à étendre les fonctionnalités d'un composant déjà existant.

1.1 Ajout de méthodes à un composant d'arbre

Démarche proposée en 3 étapes :

- Premier projet : **construire un programme** Delphi qui implante exactement les fonctionnalités de la nouvelle procédure (nouvelle action) et le tester.
- Deuxième projet : **construire une nouvelle classe** héritant du composant visuel existant, ajouter la nouvelle procédure qui vient d'être testée comme une méthode de la classe. Construire un programme de test de cette classe.
- Troisième projet : **transformer la classe en un composant**, l'installer dans la palette des composants, puis construire un programme de test du composant. Il suffira pour le programme de test de reprendre l'essentiel du programme de test de la classe.

Exemple : un composant d'arbre dérivé du TOutline



Nous avons choisi ce composant pour deux raisons essentielles :

> 1° Sur le plan pédagogique c'est un bon outil simple de visualisation des structures d'arbres. Le lecteur pourra donc en suivant la démarche proposée ajouter ses propres méthodes de parcours d'arbre de tri etc...
>
> 2° Ce composant est présent dans toutes les versions de Delphi, ce qui le désigne comme candidat idéal à la dérivation pour nous (il est amélioré depuis par un composant TtreeView plus performant et moins simple à mettre en œuvre pour un débutant)

Fonctionnalité d'affichage de tout un niveau avec une méthode

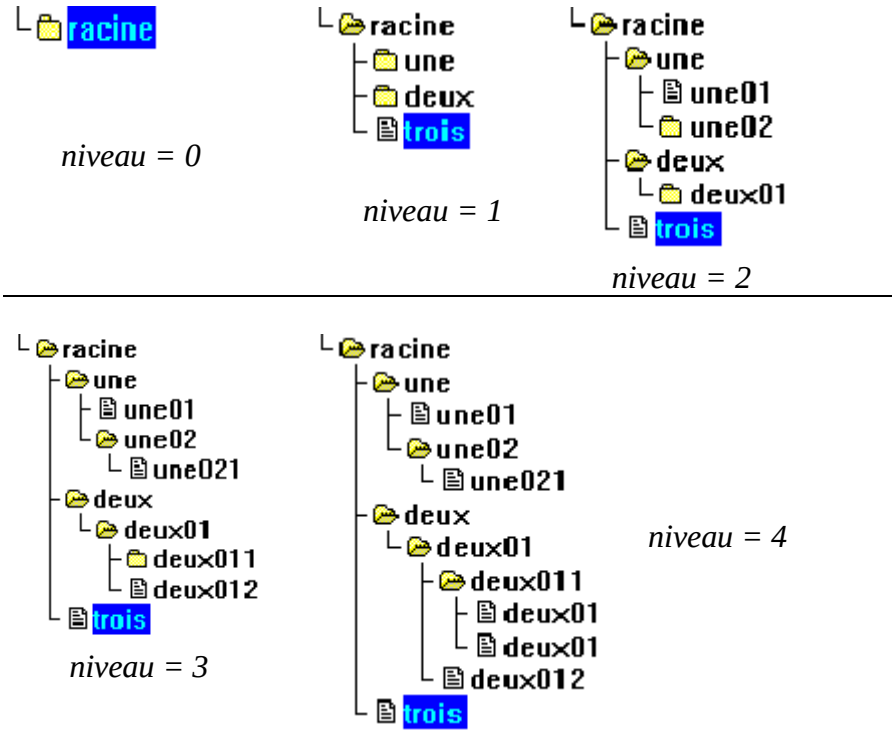
Nous désirons que notre composant affiche toutes les branches d'un arbre quelconque jusqu'à un niveau de profondeur donné. Appliquons la démarche précédente.

Projet méthode - étape/1 : *le programme Delphi*

Nous écrivons 3 méthodes Delphi permettant d'effectuer cette action :

procedure affiche_racine (tree:Toutline)	*{remonte à la racine d'où que l'on soit}*
procedure lire_un_niveau (rac:Toutline;indice:integer; niveau:integer);	*{descente recursive en préordre sur un outline}*
procedure affiche_un_niveau (le_niveau:integer);	*{ visualise tout le niveau choisi par l'utilisateur, utilise les 2 méthodes précédentes, elle sera donc public }*

C'est la procédure affiche_un_niveau qui est appelée afin d'afficher l'arbre jusqu'au niveau voulu :



Projet méthode - étape/2 : *la classe Delphi*

On crée une classe Ttree2 dérivée du Toutline

```
TTree2 = class(TOutline) {la nouvelle classe TTree dérivée de Toutline}
private
  { Déclarations private }
  procedure lire_un_niveau (rac:Toutline; indice:integer; niveau:integer);
  procedure affiche_racine (tree:Toutline);
public
 { Déclarations public }
  procedure affiche_un_niveau (le_niveau:integer);
end;
```

On déclare un objet de classe Ttree2.

```
var
  new_compos : TTree2; {objet Ttree2 déclaré}
```

On instancie l'objet de classe Ttree2.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  new_compos := TTree2.create(self); {objet Ttree2 créé}
  new_compos.parent:=self; {objet Ttree2 affichable}
  new_compos.setbounds(8,8,233,233)
end;
```

On appelle la méthode public de l'objet.

```
new_compos.affiche_un_niveau (3); {demande d' affichage niveau = 3}
```

Projet méthode - étape/3 : *le composant Delphi*

On ajoute un constructeur à la classe précédente Ttree2 :

```
TTree2 = class(TOutline) {la nouvelle classe TTree dérivée de Toutline}
private
   procedure lire_un_niveau (rac:Toutline; indice:integer; niveau:integer);
   procedure affiche_racine (tree:Toutline);
public
 constructor Create (Aowner:Tcomponent); override;
 procedure affiche_un_niveau(le_niveau:integer);
end;
```

Dans le constructeur nous reprenons le code du gestionnaire Oncreate de la fiche.

```
constructor TTree2.Create(Aowner:Tcomponent);
begin
   inherited create(Aowner);
   self.setbounds(8,8,233,233); {position : left,top,width,height}
end;
```

On respecte le mode d'enregistrement des composants en Delphi

```
procedure register;

implementation

procedure register;
begin
   RegisterComponents('Perso', [TTree2])
end;
```

Nom de l'onglet de la palette dans lequel sera rangée la classe Ttree2.

Le composant sera installé ici, dans l'onglet 'Perso'

Nom de la classe : Ttree2

Code complet du composant Ttree2

```
unit Utree2; { Un composant d'affichage d'arbre }
interface

 uses
  SysUtils, Messages, Classes, Graphics, Controls,  Forms, StdCtrls, ExtCtrls, Outline;

 type
  TTree2 = class(TOutline)
 private
  procedure affiche_racine(tree:TTree2);
  procedure TTree2.lire_un_niveau(rac:TTree2;  indice:integer;   le_niveau : integer);
 public
  constructor Create(Aowner:Tcomponent);override;
  procedure affiche_un_niveau (le_niveau : integer);
end;

  procedure register;

implementation

 procedure register;
 begin
  RegisterComponents('Perso',[TTree2])
 end;
 {---------------- Méthodes privées de la classe TTree2    ----------------}

 procedure TTree2.affiche_racine(tree:TTree2);
 {remonte à la racine d'où que l'on soit }
 var num_lev:integer;
```

```
begin
 if tree.Itemcount<>0 then
 begin
  num_lev:=tree.items[tree.selecteditem].topItem;
  tree.SelectedItem:=tree.items[num_lev].parent.index;
  tree.items[1].expanded:=false;
 end
end;

procedure TTree2.lire_un_niveau(rac:TTree2;indice:integer;le_niveau:integer);
{descente recursive en préordre sur un outline }
var
 node:ToutlineNode; {pour simplifier les manipulations}
 indice_node_fils,indice_node_pere:integer;
 begin
  if (indice<>-1)and(rac.Itemcount<>0) then
  begin
   node:=rac.items[indice];
   indice_node_pere:=rac.items[indice].parent.index;
   indice_node_fils:=indice;
   if node.HasItems then  {il y a des descendants}
   begin
    if node.level<=le_niveau then {uniquement si le niveau est correct}
    begin
     node.expand;            {visualiser les descendants à level+1}
     indice_node_pere:= rac.SelectedItem; {le noeud est le père}
     rac.SelectedItem:=rac.Items[rac.SelectedItem].GetFirstChild; {le 1er à gauche}
     indice_node_fils:=rac.SelectedItem;  {indice du noeud fils gauche}
     if indice_node_fils<>-1 then
     begin
      lire_un_niveau(rac,indice_node_fils,le_niveau);
      indice:=rac.Items[indice_node_pere].GetNextChild(indice_node_fils);{indice du frère suivant}
     end;
     while indice<>-1 do {examen de tous les frères de indice_node_fils}
     begin
      rac.SelectedItem:=indice;          {le frère suivant}
      indice_node_fils:=rac.SelectedItem; {le frère suivant est le nouveau fils}
      lire_un_niveau(rac,indice_node_fils,le_niveau);
      indice:=rac.Items[indice_node_pere].GetNextChild(indice_node_fils); {indice du frère suivant}
     end
    end
   end
  end;
 end;

{-------------------- Méthode public de la classe TTree2 ---------------------}
 procedure TTree2.affiche_un_niveau(le_niveau:integer);
 {pour visualiser tout le niveau choisi par l'utilisateur. }
 var
  indice_noeud:integer;
  begin
   affiche_racine(self);
   if le_niveau<>0 then
   begin
    indice_noeud:= self.selecteditem;
    lire_un_niveau(self,indice_noeud,le_niveau);
    if self.Itemcount<>0 then
    begin
     indice_noeud:=self.Items[1].GetNextChild(indice_noeud);
     while indice_noeud<>-1 do
```

```
    begin
     self.selecteditem:=indice_noeud;
     lire_un_niveau(self,indice_noeud,le_niveau);
     indice_noeud:=self.Items[1].GetNextChild(indice_noeud);
    end
   end
  end
 end;

{////////////////////////  CONSTRUCTEUR  ////////////////////////////}
 constructor TTree2.Create(Aowner:Tcomponent);
 begin
  inherited create(Aowner);
  self.setbounds(8,8,233,233); {position : left,top,width,height}
 end;

end.
```

1.2 Ajout de propriétés au composant d'arbre

Démarche conseillée : ***semblable à la précédente***

- Premier projet : **construire un programme** Delphi qui implante exactement les fonctionnalités de la nouvelle procédure (nouvelle fonctionnalité) et le tester.
- Deuxième projet : **construire une nouvelle classe** héritant du composant visuel existant, et ajouter la nouvelle procédure qui vient d'être testée en la reliant à une propriété publique par exemple. Construire un programme de test de cette classe avec sa nouvelle propriété.
- Troisième projet : **transformer la classe en un composant**, l'installer dans la palette des composants, puis construire un programme de test du composant. Le composant a pratiquement été entièrement construit dans l'étape précédente.

Fonctionnalité d'affichage de tout un niveau avec une propriété

Nous désirons que notre composant affiche toutes les branches d'un arbre quelconque jusqu'à un niveau de profondeur donné à partir d'une propriété Delphi et non plus d'une méthode. Nous allons ainsi voir la puissance et la facilité fournies par le RAD.

Les propriétés ont en Delphi une particularité intéressante : celle de pouvoir être **lues** ou **écrites** à travers des méthodes internes que l'on peut programmer soi-même. Ceci nous donne une latitude importante lorsque nous voulons étendre les fonctionnalités d'une classe.

L'étape/1 est strictement la même que dans le traitement précédent sur l'ajout d'une nouvelle méthode :

procedure affiche_racine (tree:Toutline)	*{remonte à la racine d'où que l'on soit}*
procedure lire_un_niveau (rac:Toutline;indice:integer; niveau:integer);	*{descente recursive en préordre sur un outline}*
procedure affiche_un_niveau (le_niveau:integer);	*{ visualise tout le niveau choisi par l'utilisateur}*

Projet propriété - étape/2 : *la classe Delphi*

On crée une classe Ttree1 dérivée du Toutline.

```
TTree1 = class(TOutline) {la nouvelle classe Ttree1 dérivée de Toutline}
private
   Fniveau : integer;
   function Getmaxniveau : integer;
   procedure affiche_un_niveau(le_niveau:integer);
   procedure lire_un_niveau (rac:Toutline; indice:integer; niveau:integer);
   procedure affiche_racine (tree:Toutline);
public
   property profondeur : integer read Getmaxniveau;
   property show_niveau  :integer read Fniveau  write affiche_un_niveau;
end;
```

La classe possède deux propriétés :

- **property** profondeur
- **property** show_niveau

Etudions l'implantation de chacune d'elle :

property profondeur :

> Afin de montrer au lecteur les possibilités de lecture et d'écriture des propriétés, nous avons rajouté à la classe une nouvelle fonctionnalité : pouvoir consulter pour un arbre donné, la valeur de sa profondeur. Cette action est implantée à travers la propriété profondeur qui est en lecture seulement (puisqu'elle est uniquement consultable).

```
property profondeur : integer read Getmaxniveau;
```

La propriété profondeur lorsqu'elle sera lue lors de l'exécution fera appel à la méthode Getmaxniveau. Cette méthode doit être une fonction et doit renvoyer un résultat du même type que la propriété (ici un integer). La méthode interne Getmaxniveau renvoyant la profondeur de l'arbre, est construite par nos soins.

Voici un exemple de code possible :

```
function TTree1.Getmaxniveau:integer;
var i,max:integer;
begin
  if self.Itemcount < 0 then begin
   max:=1;
   for i:=1 to self.Itemcount do
     if max<self.Items[i].level then
        max:=self.Items[i].level;
    Getmaxniveau:=max-1;
  end
  else Getmaxniveau:=0 { profondeur racine=0}
end;
```

Par exemple, lorsqu'une instruction du genre 'x := profondeur' est exécutée, Delphi appellera la méthode Getmaxniveau qui fournira un résultat. Ce résultat sera lui-même automatiquement placé dans la variable x. Comme nous l'avons déjà vu, les propriétés ont la particularité d'être des champs que l'on peut lire à travers une fonction.

Cette particularité est intéressante puisque nous voyons dans l'exemple que c'est au moment où l'on appelle Getmaxniveau que le calcul de la profondeur est effectué. Il s'agit d'une action dynamique qui nous assure quelle que soit la modification apportée à l'arbre, nous aurons toujours sa profondeur réelle.

Les propriétés en Delphi sont donc comme des médias permettant d'accéder à des informations à travers elles.

property show_niveau

A titre d'exemple cette propriété est en lecture et écriture.
En écriture, elle est chargée de provoquer l'affichage de l'arbre sur tout un niveau fixé.
En lecture elle fournit la valeur du niveau de l'arbre actuellement affiché.

property show_niveau : integer **read** Fniveau **write** Affiche _un_niveau;

Elle possède la particularité d'être écrite à travers une méthode qui doit être en Delphi une procédure avec un seul paramètre transmis par adresse ou par valeur mais du même type que la propriété.

- Lorsque la propriété **show_niveau** est modifiée (donc en écriture) comme dans l'instruction "*show_niveau := x*", il est fait appel à la méthode interne *Affiche_un_niveau* à laquelle Delphi passe le paramètre x. Tout se passe comme si on avait écrit "*Affiche_un_niveau(x) ;"* . Nous avons donc un moyen de provoquer dynamiquement une action lorsque l'on modifie une propriété.

- Lorsque la propriété **show_niveau** est lue comme dans l'instruction "*x := show_niveau;*" , c'est en fait le contenu du champ privé Fniveau qui est lu et renvoyé dans la variable x. Tout se passe comme si on avait écrit " *x := Fniveau ;"*

Le champ privé Fniveau contient la valeur effective du numéro de niveau qui actuellement affiché par le Ttree1, cette valeur est mise à jour lorsqu'un changement d'affichage a lieu, en l'occurrence lors de l'appel de la méthode Affiche _un_niveau(x) qui devra assurer la transmission de la valeur x dans Fniveau.

On instancie et l'on crée l'objet de classe Ttree1

```
var
  new_compos : TTree1; {objet Ttree1 déclaré}
.....
//On programme le code du gestionnaire de création de la fiche.
procedure TForm1.FormCreate(Sender: TObject);
begin
  new_compos:=TTree1.create(self); {objet TTree1 créé}
  new_compos.parent:=self; {objet TTree1 affichable}
  new_compos.setbounds(8,8,233,233); {position : left,top,width,height}
  new_compos.lines.loadfromfile('lines.txt'); {arbre chargé}
end;
```

Projet propriété - étape/3 : *le composant Delphi*

On ajoute un constructeur à la classe précédente Ttree1 :

```
TTree1 = class(TOutline) {la classe Ttree1 dérivée de Toutline}
private
   Fniveau:integer;
   function Getmaxniveau : integer;
   procedure affiche_un_niveau(le_niveau:integer);
   procedure lire_un_niveau (rac:Toutline; indice:integer; niveau:integer);
     procedure affiche_racine (tree:Toutline);
public
  constructor Create(Aowner:Tcomponent);override;
  property profondeur : integer read Getmaxniveau;
  property show_niveau : integer read Fniveau  write affiche_un_niveau;
end;
```

Identiquement à l'exemple précédent, nous remplaçons le code du gestionnaire de création de la fiche par le code du constructeur d'objets de la classe(Create).

```
constructor TTree1.Create(Aowner:Tcomponent);
begin
  inherited create(Aowner);
  self.setbounds(8,8,233,233); {position : left,top,width,height}
end;
```

On respecte le mode d'enregistrement des composants en Delphi

```
        procedure register;

        implementation

        procedure register;
        begin
         RegisterComponents('Perso', [TTree1])
        end;
```

Le composant sera installé dans l'onglet 'Perso'à côté de Ttree2 :

Code complet du composant Ttree1

```
unit Utree1; { Un composant d'affichage d'arbre }
interface

 uses
  SysUtils, Messages, Classes, Graphics, Controls,
  Forms, StdCtrls, ExtCtrls, Outline;
```

```
type
 TTree1 = class(TOutline)  {la nouvelle classe Ttree1 de Toutline}
private
 Fniveau : integer;
 function Getmaxniveau:integer;
 procedure affiche_un_niveau(le_niveau:integer);
 function Getmaxniveau : integer;
 procedure affiche_un_niveau(le_niveau:integer);
 procedure lire_un_niveau (rac:Toutline; indice:integer; niveau:integer);
 procedure affiche_racine (tree:Toutline);
public
 constructor Create(Aowner:Tcomponent);override;
  property profondeur : integer read Getmaxniveau;
  property show_niveau : integer read Fniveau write affiche_un_niveau;
end;

 procedure register;

implementation

 procedure register;
 begin
  RegisterComponents('Perso',[TTree1])
 end;

 {-------------------- méthodes private du TTree1 -----------------------}

 procedure TTree1.affiche_racine(tree:TTree1);
 {remonte à la racine d'où que l'on soit }
 var num_lev:integer;
 begin
  if tree.Itemcount<>0 then
  begin
   num_lev:=tree.items[tree.selecteditem].topItem;
   tree.SelectedItem:=tree.items[num_lev].parent.index;
   tree.items[1].expanded:=false;
  end
 end;

 procedure TTree1.lire_un_niveau(rac:TTree1;indice:integer;le_niveau:integer);
 {descente recursive en préordre sur un outline}
 var
  node:ToutlineNode; {pour simplifier les manipulations}
  indice_node_fils,indice_node_pere:integer;
  begin
   if (indice<>-1)and(rac.Itemcount<>0) then
   begin
    node:=rac.items[indice];
    indice_node_pere:=rac.items[indice].parent.index;
    indice_node_fils:=indice;
    if node.HasItems then  {il y a des descendants}
    begin
     if node.level<=le_niveau then {uniquement si le niveau est correct}
     begin
      node.expand;            {visualiser les descendants à level+1}
      indice_node_pere:= rac.SelectedItem; {le noeud est le père}
      rac.SelectedItem:=rac.Items[rac.SelectedItem].GetFirstChild; {le 1er à gauche}
      indice_node_fils:=rac.SelectedItem;  {indice du noeud fils gauche}
      if indice_node_fils<>-1 then
```

```
    begin
     lire_un_niveau(rac,indice_node_fils,le_niveau);
     indice:=rac.Items[indice_node_pere].GetNextChild(indice_node_fils);{indice du frère suivant}
    end;
    while indice<>-1 do {examen de tous les frères de indice_node_fils}
    begin
     rac.SelectedItem:=indice;          {le frère suivant}
     indice_node_fils:=rac.SelectedItem; {le frère suivant est le nouveau fils}
     lire_un_niveau(rac,indice_node_fils,le_niveau);
     indice:=rac.Items[indice_node_pere].GetNextChild(indice_node_fils); {indice du frère
suivant}
    end
   end
  end
 end;
end;

procedure TTree1.affiche_un_niveau(le_niveau:integer);
{pour visualiser tout le niveau choisi par }
var
 indice_noeud:integer;
 begin
  affiche_racine(self);
  if le_niveau<>0 then
  begin
   indice_noeud:= self.selecteditem;
   lire_un_niveau(self,indice_noeud,le_niveau);
   if self.Itemcount<>0 then
   begin
    indice_noeud:=self.Items[1].GetNextChild(indice_noeud);
    while indice_noeud<>-1 do
    begin
     self.selecteditem:=indice_noeud;
     lire_un_niveau(self,indice_noeud,le_niveau);
     indice_noeud:=self.Items[1].GetNextChild(indice_noeud);
    end
   end
  end
 end;

function TTree1.Getmaxniveau:integer;
{donne la profondeur maximum de l'arbre }
var i,max:integer;
begin
 if self.Itemcount<>0 then
 begin
  max:=1;
  for i:=1 to self.Itemcount do
   if max<self.Items[i].level then
    max:=self.Items[i].level;
  Getmaxniveau:=max-1;
 end
 else
  getmaxniveau:=0   // profondeur racine=0
end;

{////////////////////////// CONSTRUCTEUR ////////////////////////////}
constructor TTree1.Create(Aowner:Tcomponent);
{remplace le create dans l'étape 1}
begin
```

```
 inherited create(Aowner);
 self.setbounds(8,8,233,233); {position : left,top,width,height}
end;

end.
```

2. Construire par association de composants visuels

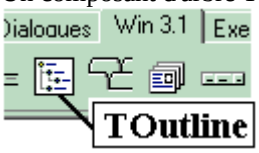
Nous nous proposons ici de fournir trois exemples de composants visuels construits par association (agrégation) d'autres composants visuels.

- Dans le premier exemple, nous montrons au lecteur comment associer trois composants visuels existants pour n'en former qu'un seul à partir du composant d'arbre fondé sur le TOutLine construit au paragraphe précédent.
- Dans le second exemple nous reprendrons la même démarche en construisant un composant plus élaboré de saisie des expressions arithmétiques en y incluant des résultats provenant des chapitres sur les analyseurs.
- le troisième exemple situé au paragraphe suivant, constitue la base de départ d'un composant d'éditeur-déboggeur d'un composant non visuel de liste de chaînes, à compléter à titre d'exercice.

2.1 Le composant de visualisation d'arbre

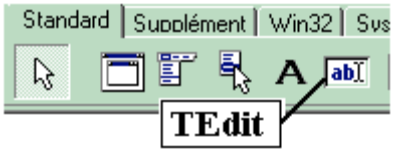
Nous montrons au lecteur comment associer trois composants visuels existants pour n'en former qu'un seul que nous notons ***TwinArbre***. Nous reprenons comme base le composant standard d'arbre, le **Toutline,** que nous associons à deux autres :

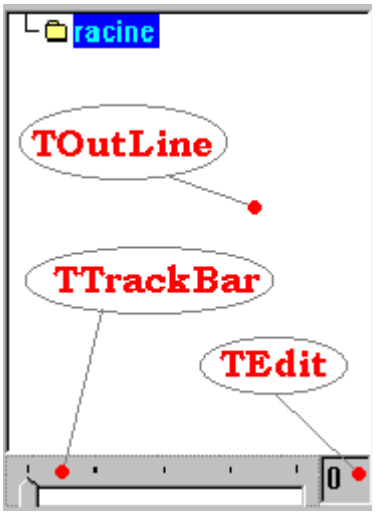
Un composant d'arbre Toutline



Un composant **TTrackBar** (barre graduée et glissière), que l'on trouve dans Win32 depuis Delphi 3

Un composant **TEdit** que l'on trouve dans l'onglet standard dans toutes les versions.

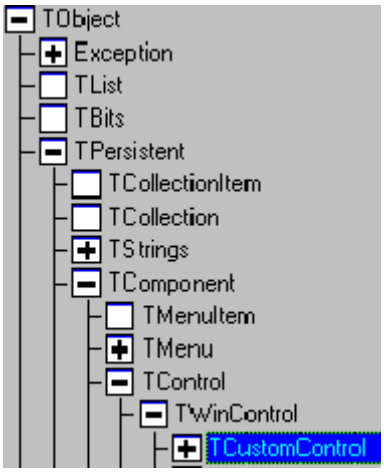




Aspect du composant ***TwinArbre*** construit par association des 3 composants de gauche

La démarche reste fondamentalement la même que celle que nous avons utilisée pour les deux exemples précédents. Ceci nous permet d'avoir une base simple, suffisante en initiation, d'élaboration de nouveaux composants.

Lorsque nous voulons construire un tel assemblage de composants il nous faut remonter dans la hiérarchie des classes. Delphi nous conseille de faire dériver notre futur composant ***TwinArbre*** systématiquement de la classe **TCustomControl** :



Nous vous livrons ci-après le composant ***TwinArbre*** en utilisant la version adjonction des propriétés. Nous élargissons les fonctionnalités par des nouvelles propriétés qui encapsulent des propriétés des 3 composants associés.

Propriétés du TwinArbre	Composant auquel elle est liée
property Potentiometre: TTrackBar **read** FPotentiometre;	La **property** Potentiometre est liée directement en lecture au composant TTrackBar.
property Tree: Toutline **read** FTree **write** FTree;	La **property** Tree est liée directement en lecture et en écriture au composant TOutline.
property Profondeur:integer **read** GetProfondeur;	La **property** profondeur est liée au composant TOutline.
property Enabled:boolean **read** Getenabled **write** Setenabled;	La **property** Enabled est reliée aux trois composants.
property Lignes: Tstrings **read** GetLignes **write** SetLignes;	La **property** Lignes est reliée au composant TOutline.
property Couleur: TColor **read** GetCouleur **write** SetCouleur;	La **property** Couleur est reliée au composant TOutline.
property Ascenceur: TScrollStyle **read** GetAscenceur **write** SetAscenceur ;	La **property** Ascenceur est reliée au composant TOutline.

Afin de donner une vue un peu plus élargie de la construction de tels composants, nous avons programmé deux événements dans notre composant ***TwinArbre*** : un événement associé au Toutline et un événement associé au TTrackBar. Nous "exportons" l'interception événementielle d'un des 3 composant associé comme étant un nouvel événement du composant **TCustomControl** formé par l'agrégation des 3 composants. La classe **TcustomControl** sert ainsi de classe "enveloppe".

Ci-dessous les codes sources de chaque propriété :

Propriétés du TwinArbre	Code Delphi de lecture/écriture

property Profondeur:integer
read GetProfondeur;
	La **property** profondeur est reliée au composant Toutline :
	function TwinArbre.GetProfondeur:integer;
	// met la profondeur de l'arbre dans la propriété profondeur
	begin
	result:=Getmaxniveau *// même métode que dans Ttree1*
	end;

property Enabled:boolean **read** Getenabled
write Setenabled;
	La **property** Enabled est reliée aux trois composants :
	function TwinArbre.GetEnabled:boolean;
	// en lecture enabled:Tboolean
	begin
	result:=FTree.enabled **and** FPotentiometre.enabled ;
	end;
	
	procedure TwinArbre.SetEnabled(x:boolean);
	// en écriture enabled:Tboolean
	begin
	FTree.Enabled:=x;
	FPotentiometre.enabled:=x ;
	if x=false **then**
	begin
	OldColor:=Ftree.color;
	Ftree.color:=clsilver;
	FEdit1.color:=clsilver
	end
	else
	begin
	Ftree.color:=OldColor;
	FEdit1.color:=OldColor
	end
	end;

property Lignes: Tstrings **read** GetLignes
write SetLignes;
	La **property** Lignes est reliée au composant Toutline:
	
	function TwinArbre.GetLignes:Tstrings;
	// en lecture lines:Tstrings
	begin
	result:=FTree.lines
	end;
	
	procedure TwinArbre.SetLignes(x:Tstrings);
	// en écriture lines:Tstrings
	begin
	FTree.lines:=x
	end;

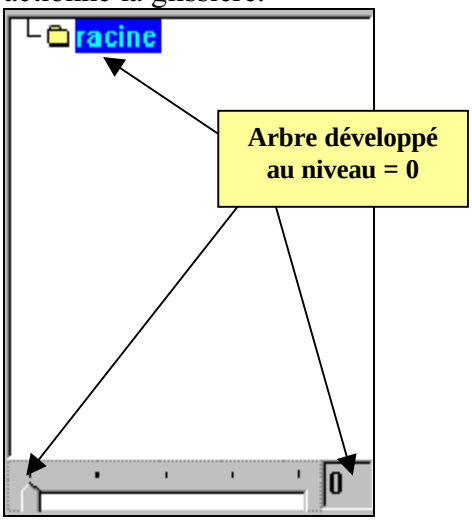
property Couleur: TColor **read** GetCouleur
write SetCouleur;
	La **property** Couleur est reliée au composant Toutline :
	function TwinArbre.GetCouleur:TColor;
	// en lecture color:TColor
	begin
	result:=FTree.color
	end;
	
	procedure TwinArbre.SetCouleur(x:TColor);
	// en écriture color:TColor
	begin
	FTree.color:=x
	end;

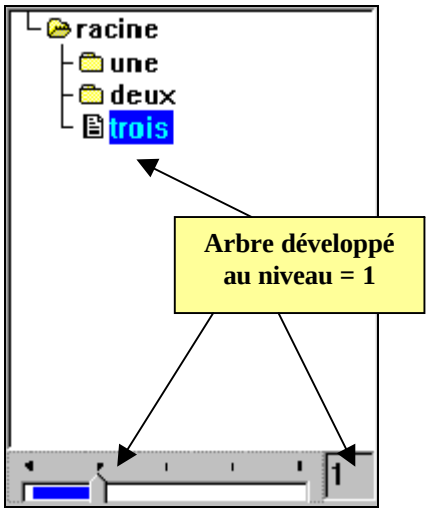
property Ascenceur: TScrollStyle **read** GetAscenceur **write** SetAscenceur ;	La **property** Ascenceur est reliée au composant Toutline :
	
	function TwinArbre.GetAscenceur:TScrollStyle;
	// en lecture ScrollBars:TScrollStyle
	begin
	result:=FTree.ScrollBars
	end;
	
	procedure TwinArbre.SetAscenceur (x:TScrollStyle);
	// en écriture ScrollBars:TScrollStyle
	begin
	FTree.ScrollBars:=x
	end;

Le lecteur pourra s'inspirer de ce développement pour écrire son code personnel sur d'autres événements.

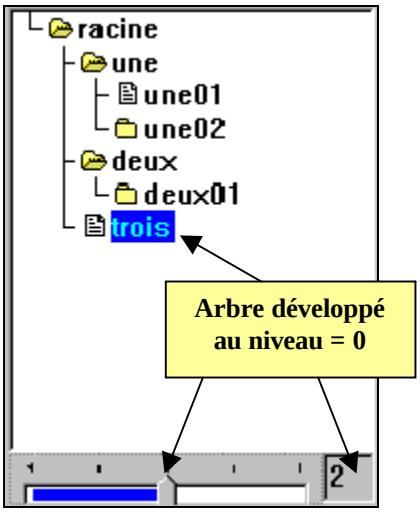
2.2 Evénement OnChange de TWinArbre

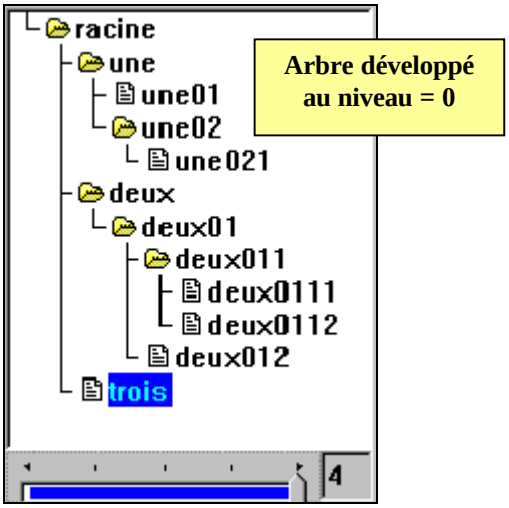
Nous avons programmé la réaction de notre composant à la manipulation de la glissière sur la barre graduée. Nous avons choisi l'événement **OnChange** du TTrackBar.
Cet événement permet d'afficher tout un niveau de l'arbre du Toutline lorsque l'utilisateur actionne la glissière.





La valeur du niveau est affichée dans le Tedit.





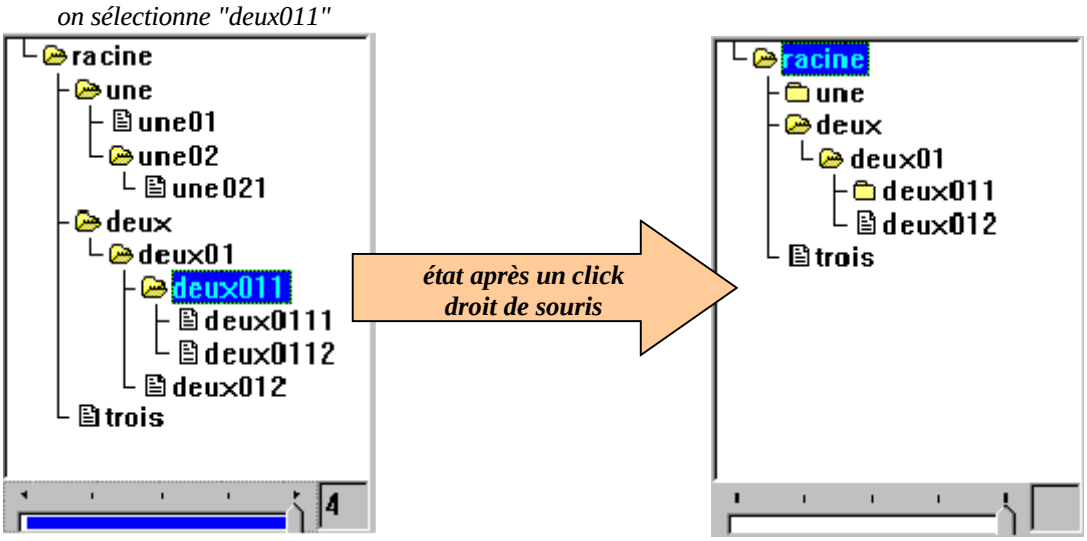
Le gestionnaire d'événement associé au niveau du **TCustomControl** doit avoir l'en-tête obligatoire d'un gestionnaire d'événement OnChange (un TNotifyEvent). Il a été nommé *PotentiometreChange*.Voici son en-tête :

```
procedure PotentiometreChange(Sender: TObject);
```

2.3 Evénement OnMouseDown de TWinArbre

Nous avons programmé de la même façon, une réaction spécifique de notre composant **TCustomControl** sur un click ***du bouton droit*** de la souris dans la zone du TOutline. Nous avons choisi pour ce faire l'événement **OnMouseDown** du Toutline.

Cet événement permet d'afficher ***seulement le chemin partant de la racine vers la feuille ou le noeud sélectionné*** ', le reste de l'arbre n'étant pas développé :



Le chemin **racine\deux\deux01\deux011** est affiché visuellement, toutes les autres branches inutiles visuellement sont refermées.

Le gestionnaire d'événement associé au niveau du **TCustomControl** a été nommé *ArbreMouseDown*. Il doit avoir l'en-tête obligatoire d'un gestionnaire d'événement **OnMouseDown** (un TmouseEvent). Voici son en-tête :

```
procedure ArbreMouseDown (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
```

2.4. Le composant TWinArbre peut se redimensionner

En anticipant sur le prochain chapitre, nous avons rajouté, à l'intention du lecteur désireux de pouvoir modifier la taille de son composant lors de la conception avec son environnement Delphi, une méthode interne spécifique de redimensionnement. Les explications complètes du

fonctionnement, se trouvent au chapitre sur les messages Windows. Nous livrons tel quel le code de la méthode autorisant le redimensionnement du composant ***TwinArbre*** :

```
private
 procedure WMSize(var Message:TWMsize); message WM_SIZE;
 ......
procedure TwinArbre.WMSize(var Message:TWMsize);
{permet de modifier la taille du TCustomcontrol lors de la conception}
begin
   inherited;
   Ftree.setbounds(0,0,width,height-25);
   FPotentiometre.setbounds(0,FTree.Top+FTree.Height,width-25,25);
   FEdit1.setbounds(FPotentiometre.left+FPotentiometre.width,  Ftree.Top+Ftree.Height,25,25);
end;
```

Sans vouloir trop entrer dans de la technicité inutile et spécifique à ce RAD, indiquons que nous avons intercepté le message de modification de taille. Nous avons conçu tous les composants en positionnement relatif les uns par rapport aux autres en prenant comme référence de départ le Toutline Ftree. L'écriture " Ftree.setbounds(0,0,width,height-25) " indique que le Toutline est positionné en top=0, left=0 du composant, qu'il a toute la largeur du composant (Width) et que sa hauteur est celle du composant moins 25 pixels (height-25). Il faut donc dessiner sur le papier soigneusement le composant avant de l'implanter.

En fait cette attitude permet d'avoir une sorte d'homothétie sur les différents éléments visuels de ***TwinArbre***. Le lecteur pourra se servir de ce composant, il possède alors, une base de travail à enrichir soit par de nouvelles propriétés, soit par des réactions à de nouveaux événements.

Code complet du composant TWinArbre

```
unit UWinArbre;

interface
 uses
  SysUtils,WinTypes,WinProcs,Messages,Classes,Graphics,Controls,
  Forms, Dialogs, StdCtrls,ExtCtrls,Grids,Outline,ComCtrls;
 type

  TwinArbre = class(TCustomControl)
 private
 FPotentiometre: TTrackBar;
 FEdit1: TEdit;
 FTree:Toutline;
 OldColor:TColor;
  procedure WMSize(var Message:TWMsize);message WM_SIZE;
  function  Getmaxniveau:integer;
  function  GetProfondeur:integer;
  function  GetEnabled:boolean;
  procedure SetEnabled(x:boolean);
  function  GetLignes:Tstrings;
  procedure SetLignes(x:Tstrings);
  function  GetCouleur:TColor;
  procedure SetCouleur(x:TColor);
```

```
  function  GetAscenceur:TScrollStyle;
  procedure SetAscenceur(x:TScrollStyle);
  procedure affiche_un_niveau(le_niveau:integer);
  procedure affiche_racine;
  procedure lire_un_niveau(indice:integer;niveau:integer);
  procedure CalCulChemin(Noeud:ToutLineNode;Liste:TStringList);
  procedure ArbreMouseDown(Sender: TObject; Button: TMouseButton;
   shift: tshiftstate;
   x, y: integer);
  procedure PotentiometreChange(Sender: TObject);
 public
   Liste: TStringList;
   constructor Create(Aowner:Tcomponent);override;
   property Potentiometre: TTrackBar read FPotentiometre;
   property Tree: Toutline read FTree write FTree;
   property Profondeur:integer read GetProfondeur;
 published
  property Enabled:boolean read Getenabled write Setenabled;
  property Lignes: Tstrings read GetLignes write SetLignes;
  property Couleur: TColor read GetCouleur write SetCouleur;
  property Ascenceur: TScrollStyle read GetAscenceur write SetAscenceur;
end;

  procedure Register;

implementation

 procedure Register;
 begin
  RegisterComponents('Perso', [TwinArbre]);
 end;
 {  ////////////////////  les constructeurs ////////////////////////}
 procedure TwinArbre.WMSize(var Message:TWMsize);
 {permet de modifier la taille du customcontrol lors de la conception}
 begin
  inherited;
  Ftree.setbounds(0,0,width,height-25);
  FPotentiometre.setbounds(0,FTree.Top+FTree.Height,width-25,25);
  FEdit1.setbounds(FPotentiometre.left+FPotentiometre.width,Ftree.Top+Ftree.Height,25,25);
 end;

 constructor TwinArbre.Create(Aowner:Tcomponent);
 begin
  inherited create(Aowner);
  OldColor:=clWindow;
  setbounds(10,10,200,200);
  Ftree:=Toutline.create(self);
  Ftree.parent:=self;
  Ftree.setbounds(0,0,width,height-25);
  FTree.OnMouseDown:=ArbreMouseDown;
  Ftree.color:=OldColor;
  FPotentiometre:=TTrackBar.create(self);
  FPotentiometre.parent:=self;
  FPotentiometre.setbounds(0,FTree.Top+FTree.Height,width-25,25);
  FEdit1:=TEdit.create(self);
  FEdit1.parent:=self;
  FEdit1.color:=OldColor;
  if Getmaxniveau>0 then
   FPotentiometre.Max:=Getmaxniveau-1
  else
```

```
 fpotentiometre.max:=0;
 FPotentiometre.Min:=0;
 FPotentiometre.Position:=0;
 FPotentiometre.LineSize:=1;
 FPotentiometre.PageSize:=1;
 FPotentiometre.TickMarks:=tmTopLeft;
 FPotentiometre.OnChange:=PotentiometreChange;
 FEdit1.setbounds(FPotentiometre.left+FPotentiometre.width,Ftree.Top+Ftree.Height,25,25);
 FEdit1.text:=inttostr(FPotentiometre.Min);
 FEdit1.ReadOnly:=true;
 Liste:=TStringList.create;
end;
{////////////////////////// implantation ////////////////////////////}

function TwinArbre.Getmaxniveau:integer;
{donne la profondeur maximum}
var i,max:integer;
begin
 if Ftree.Itemcount<>0 then
 begin
  max:=1;
  for i:=1 to Ftree.Itemcount do
   if max<Ftree.Items[i].level then
    max:=Ftree.Items[i].level;
  Getmaxniveau:=max;
 end
 else
  getmaxniveau:=0
end;

procedure TwinArbre.affiche_racine;
{remonte à la racine d'où que l'on soit}
var num_lev:integer;
begin
 if Ftree.Itemcount<>0 then
 begin
  num_lev:=Ftree.items[Ftree.selecteditem].topItem;
  Ftree.SelectedItem:=Ftree.items[num_lev].parent.index;
  Ftree.items[1].expanded:=false;
 end
end;

procedure TwinArbre.lire_un_niveau(indice:integer;niveau:integer);
{descente recursive en préordre sur un outline}
var node:ToutlineNode; {pour simplifier les manipulations}
 indice_node_fils,indice_node_pere:integer;
 begin
  if (indice<>-1)and(Ftree.ItemCount<>0) then
  begin
   node:=Ftree.items[indice];
   indice_node_pere:=Ftree.items[indice].parent.index;
   indice_node_fils:=indice;
   if node.HasItems then  {il y a des descendants}
   begin
    if node.level<=niveau then {uniquement si le niveau est correct}
    begin
     node.expand;            {visualiser les descendants à level+1}
     indice_node_pere:= Ftree.SelectedItem; {le noeud est le père}
     Ftree.SelectedItem:=Ftree.Items[Ftree.SelectedItem].GetFirstChild; {le 1er à gauche}
     indice_node_fils:=Ftree.SelectedItem;  {indice du noeud fils gauche}
```

```
      if indice_node_fils<>-1 then
      begin
       lire_un_niveau(indice_node_fils,niveau);
       indice:=Ftree.Items[indice_node_pere].GetNextChild(indice_node_fils);{indice du frère suivant}
      end;
      while indice<>-1 do {examen de tous les frères de indice_node_fils}
      begin
       Ftree.SelectedItem:=indice;          {le frère suivant}
       indice_node_fils:=Ftree.SelectedItem; {le frère suivant est le nouveau fils}
       lire_un_niveau(indice_node_fils,niveau);
       indice:=Ftree.Items[indice_node_pere].GetNextChild(indice_node_fils); {indice du frère suivant}
      end
     end
    end
   end
  end;

procedure TwinArbre.affiche_un_niveau(le_niveau:integer);
{pour visualiser tout le niveau choisi par l'utilisateur}
var
 indice_noeud:integer;
 begin
  affiche_racine; {affiche la racine dans tous les cas}
  if le_niveau<>0 then
  begin
   indice_noeud:= Ftree.selecteditem;
   lire_un_niveau(indice_noeud,le_niveau);
   if Ftree.Itemcount<>0 then
   begin
    indice_noeud:=Ftree.Items[1].GetNextChild(indice_noeud);
    while indice_noeud<>-1 do
    begin
     Ftree.selecteditem:=indice_noeud;
     lire_un_niveau(indice_noeud,le_niveau);
     indice_noeud:=Ftree.Items[1].GetNextChild(indice_noeud);
    end
   end
  end
 end;

procedure TwinArbre.CalCulChemin(Noeud:ToutLineNode;Liste:TStringList);
{stockage dans une Liste du chemin des numéros
de noeuds depuis la racine jusqu'à "noeud"  }
begin
 if Noeud.index<>1 then {on ne remonte pas au delà de la racine!}
 begin
  Liste.add(inttostr(Noeud.parent.index));
  CalCulChemin(Noeud.parent,Liste)
 end
end;
{---------------- Les gestionnaires d'événements -------------}

procedure TwinArbre.PotentiometreChange(Sender: TObject);
{le curseur sert à définir le n° du niveau de l'arbre à afficher}
begin
 if Getmaxniveau>0 then
  FPotentiometre.Max:=Getmaxniveau-1
 else
  fpotentiometre.max:=0;
 FEdit1.text:=inttostr(FPotentiometre.Position);
```

```
 FPotentiometre.Selstart:=0;
 FPotentiometre.Selend:=FPotentiometre.Position;
 affiche_un_niveau(FPotentiometre.Position);
end;

procedure TwinArbre.ArbreMouseDown(Sender: TObject; Button: TMouseButton;
{lorsqu'on clicke avec le bouton droit de souris l'objet émet un son
et il affiche uniquement le chemin permettant d'arriver à l'élément
qui est actuellement sélectionné dans l'arbre. }
shift: tshiftstate;
x, y: integer);
var numItem,i:integer;
begin
 with Sender as TOutLine do
 begin
  numItem:=selecteditem;
  if numItem>0 then {-1 pour rien de sélectionné et 0 si vide}
   if Button=mbRight then
   begin
    MessageBeep(MB_ICONASTERISK); {function de l'API Windows: émet un son}
    Liste.clear;
    CalCulChemin(items[numItem],Liste);
    items[1].Collapse;          {fermeture de tout l'arbre}
    for i:=Liste.count-1 downto 0 do
     Items[strtoint(Liste.strings[i])].expand; {expansion des parents seulement}
    FEdit1.text:='';            {on montre que cette partie n'est pas active}
    FPotentiometre.Selstart:=0; {    ---   idem    ---    }
    FPotentiometre.Selend:=0;   {    ---   idem    ---    }
   end
 end
end;
{////////////////////// LES PROPRIETES ////////////////////////////}

function TwinArbre.GetProfondeur:integer;
// met la profondeur de l'arbre dans la propriété profondeur
begin
 result:=Getmaxniveau
end;

{----------  les propriétés héritées des composants utilisés ---------}

function  TwinArbre.GetLignes:Tstrings;  // en lecture lines:Tstrings
begin
 result:=FTree.lines
end;

procedure  TwinArbre.SetLignes(x:Tstrings);  // en écriture lines:Tstrings
begin
 FTree.lines:=x
end;

function  TwinArbre.GetCouleur:TColor;  // en lecture color:TColor
begin
 result:=FTree.color
end;

procedure  TwinArbre.SetCouleur(x:TColor); // en écriture color:TColor
begin
 FTree.color:=x
end;
```

```
function  TwinArbre.GetAscenceur:TScrollStyle;  // en lecture ScrollBars:TScrollStyle
begin
 result:=FTree.ScrollBars
end;

procedure  TwinArbre.SetAscenceur(x:TScrollStyle);  // en écriture ScrollBars:TScrollStyle
begin
 FTree.ScrollBars:=x
end;

function  TwinArbre.GetEnabled:boolean;  // en lecture enabled:Tboolean
begin
 result:=FTree.enabled and FPotentiometre.enabled ;
end;

procedure  TwinArbre.SetEnabled(x:boolean);  // en écriture enabled:Tboolean
begin
 FTree.Enabled:=x;
 FPotentiometre.enabled:=x ;
 if x=false then
 begin
  OldColor:=Ftree.color;
  Ftree.color:=clsilver;
  FEdit1.color:=clsilver
 end
 else
 begin
  Ftree.color:=OldColor;
  FEdit1.color:=OldColor
 end
end;
end.
```

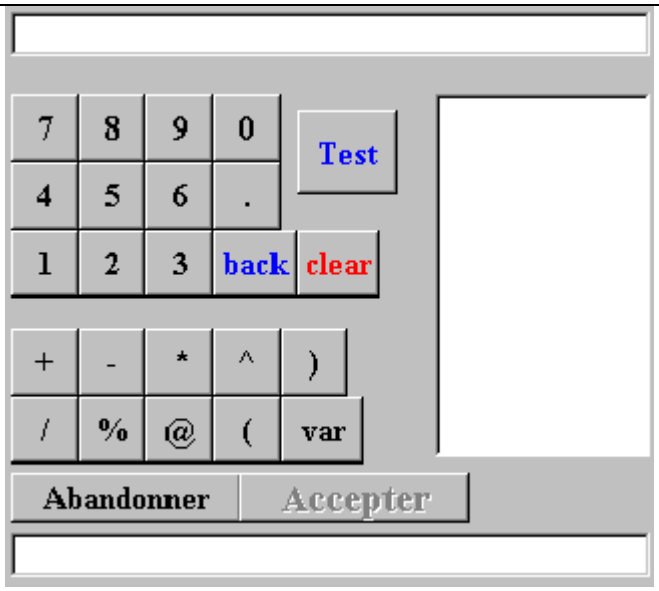
2.5 Un composant de saisie d'expression arithmétique avec Init et Follow

Voici le code d'un autre composant de saisie des expressions arithmétiques, élaboré à partir de plusieurs composants visuels associés.

Son développement met en œuvre la démarche du paragraphe précédent associée à une saisie par filtrage étudiée au chapitre de l'analyse des grammaires LL(1), avec un analyseur descendant récursif qui est inclu dans la unit du composant.

Le lecteur analysera les fonctions du logiciel et pourra y adjoindre de nouvelles fonctionnalités :

- Ajouter des boutons < , >, =, et les programmer.
- Ajouter des événements : OnTest, OnAccepter…



Code complet du composant TExpraritm

```
unit UComposExprarithm;
// composant de saisie d'expressions arithmétiques par filtrage
interface

 uses Controls,StdCtrls,Buttons,WinTypes,Classes;
 const
  et='\';
  ou='|';
  non='~';
  opdiv='%';
  opmod='@';
  Maxlongexpr=50;
  maxvar=50;
  LesChiffres='0123456789.';
  LesOper='+-*^)/%@(';
  LesCompar='<<>><=';// pour extension ultérieure
  ExprFausse='*****';
 type
  TypBouton=(T0,T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,TTest,Tback,Tclear,Tplus,
  Tmoins,Tmult,Tpuiss,Tparferm,Tdiv,Tdivint,Tmod,Tparouvr,
  Tvar,Tleave,TFin);
  TableBouton=array[TypBouton]of TBitBtn;
  enscar=set of char;

  TExpraritm=class(TCustomControl)
 private
  Edit1: TEdit;
  Edit2: TEdit;
  ListBoxvar: TListBox;
  UnBouton:TableBouton;
  numcar: integer;
  carlu: char;
  LesFollow,LesInit:enscar;
  Chiffres,Lettres:enscar;
  edit_expr,expr_loc,expr_err:string; {expressions dans l'éditeur}
  erreur_expr:boolean;{erreur dans l'expression}
  procedure ListBoxClick(Sender: TObject);
  procedure BitBtnClick(Sender: TObject);
  procedure BitBtnTestClick(Sender: TObject);
  procedure BitBtnbackClick(Sender: TObject);
  procedure BitBtnclearClick(Sender: TObject);
  procedure BitBtnvarClick(Sender: TObject);
  procedure BitBtnleaveClick(Sender: TObject);
  procedure BitBtnfinClick(Sender: TObject);
  procedure expr;
  procedure init;
  procedure carsuiv;
  procedure err(n:integer);
  function  GetLignes:Tstrings;
  procedure SetLignes(x:Tstrings);
 public
  Expression:string;
  Reconnue:boolean;
  constructor create(Aowner:Tcomponent);override;
 published
  property Lignes:TStrings read GetLignes write SetLignes;
end;
```

```
 procedure Register;

implementation
uses Dialogs,SysUtils,Graphics;

 procedure Register;
 begin
  RegisterComponents('Perso', [TExpraritm]);
 end;
 {---------------- Utilitaires de positionnement-------------------}
 procedure PositionBoutonsChiffre(var Table:TableBouton);
 const h=33;
  w=33;
 var i:TypBouton;
 begin
  Table[T7].setbounds(8,48,H,W);
  Table[T8].setbounds(41,48,H,W);
  Table[T9].setbounds(74,48,H,W);
  Table[T0].setbounds(107,48,H,W);
  Table[T4].setbounds(8,81,H,W);
  Table[T5].setbounds(41,81,H,W);
  Table[T6].setbounds(74,81,H,W);
  Table[T10].setbounds(107,81,H,W);
  Table[T1].setbounds(8,114,H,W);
  Table[T2].setbounds(41,114,H,W);
  Table[T3].setbounds(74,114,H,W);
  for i:=T0 to T10 do
   Table[i].caption:=LesChiffres[ord(i)+1];
  {----------  Les autres boutons ----------}
  Table[Tback].setbounds(107,114,41,W);
  Table[Tback].caption:='back';
  Table[Tback].Font.color:=clBlue;
  Table[Tclear].setbounds(148,114,41,W);
  Table[Tclear].caption:='clear';
  Table[Tclear].Font.color:=clRed;
  Table[TTest].setbounds(148,56,49,41);
  Table[TTest].caption:='Test';
  Table[TTest].Font.color:=clBlue;
  Table[Tleave].setbounds(8,232,113,25);
  Table[Tleave].caption:='Abandonner';
  Table[TFin].setbounds(120,232,113,25);
  Table[TFin].caption:='Accepter';
  Table[TFin].Font.color:=clRed;
  Table[TFin].Font.Size:=14;
  Table[TFin].enabled:=false;
 end;

 procedure PositionBoutonsOper(var Table:TableBouton);
 const h=33;
  w=33;
 var i:TypBouton;
 begin
  Table[Tplus].setbounds(8,162,H,W);
  Table[Tmoins].setbounds(41,162,H,W);
  Table[Tmult].setbounds(74,162,H,W);
  Table[Tpuiss].setbounds(107,162,H,W);
  Table[Tparferm].setbounds(140,162,H,W);
  Table[Tdiv].setbounds(8,195,H,W);
  Table[Tdivint].setbounds(41,195,H,W);
```

```
 Table[Tmod].setbounds(74,195,H,W);
 Table[Tparouvr].setbounds(107,195,H,W);
 Table[Tvar].setbounds(140,195,41,W);
 for i:=Tplus to Tparouvr do
  Table[i].caption:=LesOper[ord(i)-ord(Tplus)+1];
 Table[Tvar].caption:='var';
end;
{////////////////////// la construction ////////////////////}
constructor TExpraritm.create(Aowner:Tcomponent);
var num:TypBouton;
begin
 inherited create(Aowner);
 setbounds(0,0,325,290);
 for num:= T0 to TFin do
 begin
  UnBouton[num]:=TBitBtn.create(self);
  UnBouton[num].parent:=self;
  UnBouton[num].Font.name:='Times New Roman';
  UnBouton[num].Font.Style:=[fsBold];
  UnBouton[num].Font.Size:=12;
  UnBouton[num].OnClick:=BitBtnClick;
 end;
 UnBouton[TTest].OnClick:=BitBtnTestClick;
 UnBouton[Tback].OnClick:=BitBtnbackClick;
 UnBouton[Tvar].OnClick:=BitBtnvarClick;
 UnBouton[Tleave].OnClick:=BitBtnleaveClick;
 UnBouton[TFin].OnClick:=BitBtnfinClick;
 UnBouton[Tclear].OnClick:=BitBtnclearClick;
 PositionBoutonsChiffre(UnBouton);
 PositionBoutonsOper(UnBouton);
 Edit1:=TEdit.create(self);
 Edit2:=TEdit.create(self);
 Edit1.parent:=self;
 Edit1.setbounds(8,8,313,27);
 Edit1.color:=clAqua;
 Edit1.ReadOnly:=true;
 Edit2.parent:=self;
 Edit2.setbounds(8,262,313,27);
 Edit2.color:=clYellow;
 ListBoxvar:=TListBox.create(self);
 ListBoxvar.parent:=self;
 ListBoxvar.setbounds(216,48,105,177);
 ListBoxvar.OnClick:=ListBoxClick;
 Expression:=ExprFausse;
 Reconnue:=false;
end;

{////////////  Les gestionnaires d'événements Onclick ////////////}
procedure TExpraritm.ListBoxClick(Sender: TObject);
begin
 if ListBoxvar.itemIndex<>-1 then
 begin
  expr_loc:=concat(expr_loc,ListBoxvar.items[ListBoxvar.itemIndex]);
  Edit1.text:=expr_loc
 end
end;

procedure TExpraritm.BitBtnClick(Sender: TObject);
var car:char;
begin
```

```
 with sender as TBitBtn do
 begin
  UnBouton[TFin].enabled:=false;
  car:=caption[1];
  if car in ['0'..'9']+['%','/','+','-','*','\','@','(',')','.','^'] then
   expr_loc:=concat(expr_loc,caption)
 end;
 edit1.text:=expr_loc;
end;

procedure TExpraritm.BitBtnTestClick(Sender: TObject);
begin
 Expression:=ExprFausse;
 Reconnue:=false;
 if length(expr_loc)<=Maxlongexpr then
 begin
  init;
  edit2.text:='';
  edit_expr:=expr_loc;
  edit_expr := concat(edit_expr,'#');
  carsuiv;
  erreur_expr:=false;
  expr;
  if erreur_expr then
   edit2.text:=expr_err
  else
  begin // expression correcte
   UnBouton[TFin].enabled:=true;
   Expression:=expr_loc
  end
 end
 else
  MessageDlg('Expression trop longue !', mtWarning,[mbOk], 0);
end;

procedure TExpraritm.BitBtnbackClick(Sender: TObject);
begin
 expr_loc:=copy(expr_loc,1,length(edit1.text)-1);
 edit1.text:=expr_loc;
 UnBouton[TFin].enabled:=false;
end;

procedure TExpraritm.BitBtnclearClick(Sender: TObject);
begin
 expr_loc:='';
 edit1.text:=expr_loc;
 UnBouton[TFin].enabled:=false;
end;

procedure TExpraritm.BitBtnvarClick(Sender: TObject);
begin
 if UnBouton[Tvar].tag=1 then
 begin
  ListBoxvar.visible:=true;
  UnBouton[Tvar].tag:=0;
 end
 else
 begin
  ListBoxvar.visible:=false;
  UnBouton[Tvar].tag:=1;
```

```
 end
end;

procedure TExpraritm.BitBtnleaveClick(Sender: TObject);
begin
 edit_expr:=ExprFausse;
 UnBouton[TFin].enabled:=false;
 Reconnue:=false;
 MessageBeep(MB_ICONASTERISK); {function de l'API Windows}
 Edit2.text:='Ok abandon reconnu!'
end;

procedure TExpraritm.BitBtnfinClick(Sender: TObject);
begin
 Reconnue:=true;
 MessageBeep(MB_ICONQUESTION); {function de l'API Windows}
 Edit2.text:='Ok expression acceptée !'
end;
{//////////////////////    Les Propriétés   ////////////////////////}
function  TExpraritm.GetLignes:Tstrings;
begin
 result:=ListBoxvar.items
end;

procedure TExpraritm.SetLignes(x:Tstrings);
begin
 ListBoxvar.items:=x
end;

{///////////  Analyseur descendant récursif d'expressions /////////////}
procedure TExpraritm.init;
begin
 numcar := 0;
 LesFollow:=['+', '-', '*', '/','@','%',')','#','^'];
 LesInit:=['(','^','a'..'z','0'..'9'];
 Chiffres:=['0'..'9'];
 Lettres:=['a'..'z'];
end;

procedure TExpraritm.carsuiv;
Var CharS:string[1];
begin
 numcar := numcar + 1;
 CharS:=LowerCase(edit_expr[numcar]);
 carlu := CharS[1];
end;

procedure TExpraritm.err(n:integer);
var
 i: integer;
 begin
  expr_err:=concat('>>> Erreur ',inttostr(n),': ');
  Edit1.Hideselection:=false;
  Edit1.SelStart:=numcar-1; // sélection du car erroné
  Edit1.SelLength:=1;
  for i := 1 to numcar do
   expr_err:=concat(expr_err,edit_expr[i]);
  if carlu='#' then
   carlu:=edit_expr[numcar-1];
  expr_err:=concat(expr_err,' <--[ ',carlu,' ]');
```

```
 erreur_expr:=true;
end;

procedure TExpraritm.expr;
procedure fact;
begin
 if not erreur_expr then
 begin
  if carlu in LesInit then
  begin
   case
    carlu of
    '^':
    begin
     carsuiv;
     if carlu in LesInit then
     begin
      Fact;
      if erreur_expr then
       exit;
      carsuiv;
      if not(carlu in lesfollow) then
       err(7)
     end
     else
      err(8)
    end;
    '(':
    begin
     carsuiv;
     if carlu in LesInit then
     begin
      Expr;
      if erreur_expr then
       exit;
      if carlu<>')' then
       err(9)
      else
       carsuiv;
      if not(carlu in lesfollow) then
       err(10)
     end
     else
      err(11)
    end;
    'a'..'z':
    begin
     carsuiv;
     if carlu in Chiffres+Lettres then
      while carlu in Chiffres+Lettres do
       carsuiv;
     if not(carlu in lesfollow) then
      err(12)
    end;
    '0'..'9':
    begin
     carsuiv;
     while carlu in chiffres do
      carsuiv;
     if carlu='.' then
```

```
    begin
     carsuiv;
     while carlu in chiffres do
      carsuiv;
     if not(carlu in lesfollow) then
      err(13)
    end;
    if not(carlu in lesfollow) then
     err(14)
   end;
  end{case}
 end
end
end;

procedure Terme;
begin
 if not erreur_expr then
 begin
  if carlu in LesInit then
  begin
   Fact;
   if erreur_expr then
    exit;
   while carlu in['*', '/','@','%','^'] do
   begin
    carsuiv;
    if carlu in lesinit then
     fact
    else
     err(1);
    if erreur_expr then
     exit;
   end;
   if not (carlu in lesfollow) then
    err(2)
  end
  else
   err(3)
 end
end;

begin{expr}
 if not erreur_expr then
 begin
  if carlu in LesInit then
  begin
   Terme;
   if erreur_expr then
    exit;
   while carlu in['+', '-'] do
   begin
    carsuiv;
    if carlu in lesinit then
     terme
    else
     err(4);
    if erreur_expr then
     exit;
   end;
```

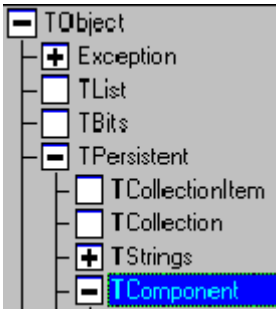
```
  if not (carlu in lesfollow) then
   err(5)
  end
  else
   err(6)
 end
end;{expr}

end.
```

3. Construire un composant non visuel

En utilisant notre démarche de construction progressive et de tests successifs, nous pouvons sans effort particulier encapsuler dans un composant non visuel beaucoup d'unités réutilisables.

Nous allons encapsuler un TStringList dans un composant non visuel *construit à partir de la classe de composant abstrait TComponent* que nous nommons Tliste.

La seule différence avec les exemples précédents réside dans la classe de départ dont nous devons faire hériter notre futur composant. En Delphi nous devons dériver notre composant non visuel de la classe **Tcomponent**. La classe **TComponent** est le point de départ abstrait de tous les composants de Delphi et nous devons remonter à ce niveau lorsque nous voulons construire un composant qui ne sera pas un contrôle (un contrôle = un composant visuel).




```
TListe = class (TComponent)
        private
            FListGeneric:TstringList;
            ....
        public
            property ListGeneric:TstringList
                                read FListGeneric
                                write FListGeneric;
            ....
        end;
```

Comme nous voulions malgré tout bénéficier des propriétés et des méthodes de la classe TStringList, nous avons utilisé la démarche suivante :

- construire un champ privé du type dont on veut dériver (ici TstringList),
- construire une propriété du composant qui permettra de lire et d'écrire dans ce champ.

L'effort de construction que nous avons à apporter est minimal puisqu'il concerne uniquement l'exportation des méthodes

Code complet du composant TListe

Les propriétés et les méthodes sont fournies à titre d'exemple pédagogique. Nous encourageons le lecteur à réécrire et à développer lui-même à partir du composant TListe un composant personnalisé de liste de chaînes.

```
unit UListeCompos;

 interface

 uses
  StdCtrls,
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

 type
  TListe = class(TComponent)
 private
  FListGeneric:TstringList;
  function  Getcount:integer;
  function  GetDuplic:boolean;
  procedure SetDuplic(x:boolean);
  function  Getsorted:boolean;
  procedure Setsorted(x:boolean);

 public
  constructor create(Aowner:Tcomponent);override;
  destructor  Liberer;
  {opérateurs de manipulation de la liste}
  procedure Effacer;
  function  Element(rang:integer):string;
  function  Position(element:string):integer;
  procedure Ajouter(ch:string);
  procedure Inserer(element:string;rang:integer);
  procedure Modifier(rang:integer;element:string);
  procedure Supprimer(rang:integer);
  {opérateurs de recopie d'une liste}
  procedure EstUneCopiede(T:TListe);
  procedure VersListBox(LBox:TListBox);
  {opérateurs d'entrée/sortie dans une liste}
  procedure Charger;
  procedure Sauver;
  {propriétés publiques, uniquement consultables}
  property Quantite:integer read Getcount ;
  property ListGeneric:TstringList read FListGeneric;

 published
  property Dupliquee:boolean read GetDuplic write SetDuplic;
  property Triee:boolean read Getsorted write Setsorted;
end;

  procedure Register;

implementation

 procedure Register;
 begin
  RegisterComponents('Perso', [TListe]);
```

```
end;

{////////////////////////   PARTIE PUBLIC   ////////////////////////////}
constructor TListe.create(Aowner:Tcomponent);
begin
 inherited create(Aowner);
 FListGeneric:=TstringList.create;
 dupliquee:=false;
 triee:=false;
end;

destructor TListe.Liberer;
begin
 FListGeneric.free;
 inherited destroy;
end;
{///////-------------------- Méthodes publiques --------------------/////}
procedure TListe.Effacer;
{ré-initialise à vide}
begin
 FListGeneric.clear;
end;

function TListe.Element(rang:integer):string;
{fournit l'élément string de n°rang}
begin
 if rang in [1..FListGeneric.count] then
  Element:=FListGeneric.strings[rang-1]
 else
 begin
  Element:='';
  Application.MessageBox('>>> Méthode : ELEMENT',
  'Rang d''élément hors limites', mb_OK);
 end
end;

function TListe.Position(element:string):integer;
{ donne le rang de la première apparition de l'élément:
   position=0 si non l'élément n'est pasprésent dans la liste }
begin
 Position:=FListGeneric.IndexOf(element)+1
end;

procedure TListe.Ajouter(ch:string);
{ajout d'un élément en fin de liste}
begin
 FListGeneric.Add(ch);
end;

procedure TListe.Inserer(element:string;rang:integer);
{insérer l'élément donné au rang indiqué}
begin
 if not Triee then
  if rang in [1..FListGeneric.count] then
   FListGeneric.Insert(rang-1,element)
  else
   Application.MessageBox('>>> Méthode : INSERER',
 'Rang d''élément hors limites', mb_OK);
end;
```

```
procedure TListe.Modifier(rang:integer;element:string);
{changer le contenu au rang indiqué par l'élément donné}
begin
 if not Triee then
  if rang in [1..FListGeneric.count] then
   FListGeneric.strings[rang-1]:=element
  else
   Application.MessageBox('>>> Méthode : MODIFIER',
 'Rang d"élément hors limites', mb_OK);
end;

procedure TListe.Supprimer(rang:integer);
{supprimer l'élément situé à ce rang}
begin
 if rang in [1..FListGeneric.count] then
  FListGeneric.Delete(rang-1)
 else
  Application.MessageBox('>>> Méthode : SUPPRIMER',
 'Rang d"élément hors limites', mb_OK);
end;
{/////------------------ recopie d'une liste----------------------/////}

procedure TListe.EstUneCopiede(T:TListe);
{effectue une copie de liste}
begin
 Effacer;
 FListGeneric.Assign(T.ListGeneric);
end;

procedure TListe.VersListBox(LBox:TListBox);
{recopie en mode string le contenu de la liste dans une ListBox}
begin
 if assigned(LBox) then
  LBox.items:=FListGeneric
end;
{/////------------------------ ENTREE/SORTIE------------------------/////}
procedure TListe.Charger;
{charger la liste à partir d'un fichier}
var Ouvrir:TOpenDialog;
begin
 Ouvrir:=TOpenDialog.create(self);
 Ouvrir.filter:='Texte|*.txt';
 Ouvrir.Title:='Chargement d"une liste';
 Ouvrir.Options:=[ofFileMustExist,ofHideReadOnly];
 if Ouvrir.execute then
  FListGeneric.LoadFromfile(Ouvrir.Filename);
 Ouvrir.free
end;

procedure TListe.Sauver;
{sauver la liste à partir dans un fichier}
var Save:TSaveDialog;
begin
 Save:=TSaveDialog.create(self);
 Save.filter:='Texte|*.txt';
 Save.Title:='Sauvegarder la liste';
 Save.Options:=[ofOverwritePrompt,ofHideReadOnly];
 if Save.execute then
  FListGeneric.SaveTofile(Save.Filename);
 Save.free
```

```
end;

{----------------------   PROPRIETE PUBLIC   -------------------------}
function TListe.Getcount:integer;
{fournit le nombre d'éléments dans la propriété Quantite}
begin
 Getcount:=FListGeneric.count
end;

{----------------------   PROPRIETES PUBLISHED   ---------------------}
function TListe.GetDuplic:boolean;
{méthode de lecture de l'autorisation d'élément dupliqué}
begin
 if FListGeneric.duplicates=dupAccept then
  GetDuplic:=true
 else
  getduplic:=false
end;

procedure TListe.SetDuplic(x:boolean);
{méthode d'écriture autorisant ou non la duplication d'éléments}
begin
 if x=true then
  FListGeneric.duplicates:=dupAccept
 else
  flistgeneric.duplicates:=duperror
end;

function TListe.Getsorted:boolean;
{méthode de lecture de l'autorisation de liste triée}
begin
 Getsorted:=FListGeneric.sorted
end;

procedure TListe.Setsorted(x:boolean);
{méthode d'écriture autorisant ou non le tri de la liste}
begin
 if x=true then
  FListGeneric.sorted:=true
 else
  flistgeneric.sorted:=false
end;

end.
```

3.2 Utilisation du composant TListe

Un exemple d'utilisation du composant liste : Il s'agit d'une interface de stockage dans trois listes de type TListe de la taille et du poids idéal d'un individu (à partir d'un exemple d'un ouvrage sur Visual basic) :

ListeIndiv : TListe
ListeHomme : TListe
ListeFemme : TListe

Au fur et à mesure de la demande de l'utilisateur, le programme range dans la liste " ListeIndiv " les informations sur la personne. Il est possible à chaque instant de construire à partir de cette liste deux sous listes selon le sexe des individus (ListeHomme et ListeFemme).

Le logiciel " PoidsListe" assure un certain niveau de sécurité en utilisant les principes de programmation défensive étudiés au chapitre correspondant.

Code source utilisant le composant TListe

```
unit UFListPoids;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls, Buttons, UListeCompos;

 type
  TForm1 = class(TForm)
  OptionFemme: TRadioButton;
  OptionHomme: TRadioButton;
  Image1: TImage;
  Image2: TImage;
  Label1: TLabel;
  SaisieTaille: TEdit;
  Bevel3: TBevel;
  AffichePoids: TLabel;
  Label5: TLabel;
  Label4: TLabel;
  Label2: TLabel;
  Bevel1: TBevel;
  Bevel2: TBevel;
  Label3: TLabel;
  BitBtnStockListe: TBitBtn;
  ListBox1: TListBox;
```

```
  BitBtnvoirListFemme: TBitBtn;
  BitBtnvoirListHomme: TBitBtn;
  BitBtnListEntiere: TBitBtn;
  Label6: TLabel;
  Label7: TLabel;
  BitBtnCreeListFemme: TBitBtn;
  BitBtnCreeListHomme: TBitBtn;
  Bevel6: TBevel;
  Bevel4: TBevel;
  BitBtnEffaceEcran: TBitBtn;
  Label8: TLabel;
  ListeIndiv: TListe;
  ListeHomme: TListe;
  ListeFemme: TListe;
  Liste1: TListe;
  procedure OptionHommeClick(Sender: TObject);
  procedure OptionFemmeClick(Sender: TObject);
  procedure SaisieTailleChange(Sender: TObject);
  procedure FormCreate(Sender: TObject);
  procedure BitBtnStockListeClick(Sender: TObject);
  procedure BitBtnCreeListFemmeClick(Sender: TObject);
  procedure BitBtnCreeListHommeClick(Sender: TObject);
  procedure BitBtnvoirListFemmeClick(Sender: TObject);
  procedure BitBtnvoirListHommeClick(Sender: TObject);
  procedure BitBtnListEntiereClick(Sender: TObject);
  procedure BitBtnEffaceEcranClick(Sender: TObject);
 private
 { Déclarations privées }
 public
 { Déclarations publiques }
end;
Tsexe=(homme,femme);
Tindividu=class
 public
  Etat:Tsexe;
  Taille:integer;
  Poids:real;
 private
  procedure CalculPoids;
  function CoherenceTaille:Boolean;
end;

 var
  Form1: TForm1;
  Personne:Tindividu;

implementation
const KFemme='Féminin';
 KHomme='Masculin';
var ChaineListe:string;

 {$R *.DFM}
 {--------------------- INITIALISATION DE LA FENETRE -------------------}
 procedure TForm1.FormCreate(Sender: TObject);
 begin
  Personne:=Tindividu.create;
 end;

 {----------------- METHODES DE LA CLASSE Tindividu --------------------}
```

voir Liste Féminin
voir Liste Masculin
voir Liste Entière
créer Liste Féminin
créer Liste Masculin

Composants de Tliste non visuels

```
procedure Tindividu.CalculPoids;
{calcule le poids si la taille est dans les limites, puis affiche le
résultat dans affichepoids }
begin
 if CoherenceTaille then
 begin
  case
   etat of
   femme:
   begin
    Poids:=(Taille-100)*0.85;
    Form1.AffichePoids.Caption:=FloatToStrF(Poids,ffFixed,5,2);
    ChaineListe:=KFemme+'  ';
   end;
   homme:
   begin
    Poids:=(Taille-100)*0.9;
    Form1.AffichePoids.Caption:=FloatToStrF(Poids,ffFixed,5,2);
    ChaineListe:=KHomme+' '
   end;
  end;{case}
  Form1.BitBtnStockListe.enabled:=true; {active le bouton de stockage dans liste}
 end
 else
 begin
  Form1.BitBtnStockListe.enabled:=false; {désactive le bouton de stockage}
  Form1.AffichePoids.caption:=''; {efface l'affichage du poids}
 end
end;{-- Fin CalculPoids --}

function Tindividu.CoherenceTaille:Boolean;
{indique si la taille est dans les limites acceptées}
begin
 if (not(Taille in [100..190])and(Etat=Femme))or
  (not(Taille in [100..210])and(Etat=Homme))then
 CoherenceTaille:=false
 else
  coherencetaille:=true
end; {-- Fin CoherenceTaille --}

{////////////////////   LES OBJETS VISUELS  ////////////////////////////}

{------------------------ LES OBJETS DE SAISIE -------------------------}
procedure TForm1.OptionHommeClick(Sender: TObject);
{permet de saisir le champ etat}
begin
 Personne.Etat:=homme;
 if SaisieTaille.Text<>'' then
  SaisieTaille.clear;
 SaisieTaille.SetFocus;{par souplesse pour l'utilisateur}
end;

procedure TForm1.OptionFemmeClick(Sender: TObject);
{permet de saisir le champ etat}
begin
 Personne.Etat:=femme;
 if SaisieTaille.Text<>'' then
  SaisieTaille.clear;
 SaisieTaille.SetFocus;{par souplesse pour l'utilisateur}
end;
```

```
procedure TForm1.SaisieTailleChange(Sender: TObject);
{permet de saisir le champ taille}
begin
 if OptionFemme.Checked or OptionHomme.Checked then
 begin
  try
   Personne.Taille:=StrtoInt(SaisieTaille.Text);
  except  {le contenu n'est pas entier ? => "EconvertError"}
   on econverterror do
   personne.taille:=0;
  end;
  Personne.CalculPoids
 end
end;

{----------------  STOCKAGE DANS LA LISTE PRINCIPALE  ------------------}

procedure TForm1.BitBtnStockListeClick(Sender: TObject);
begin
 BitBtnStockListe.enabled:=false;
 ListeIndiv.Ajouter(ChaineListe+' '+SaisieTaille.Text+'    '+AffichePoids.caption);
 ListeIndiv.VersListBox(ListBox1); {visualise sur écran la liste}
 SaisieTaille.SetFocus;          {par souplesse pour l'utilisateur}
 BitBtnListEntiere.enabled:=true;
 BitBtnCreeListHomme.enabled:=true; {active le bouton création liste Hom}
 BitBtnCreeListFemme.enabled:=true; {active le bouton création liste Fem}
 BitBtnEffaceEcran.enabled:=true;   {active le bouton effacement d'écran}
 BitBtnvoirListFemme.enabled:=false;{désactive le bouton voir liste Fem}
 BitBtnvoirListHomme.enabled:=false;{désactive le bouton voir liste Hom}
end;

{------------------------ CREATION DES LISTES ------------------------}

procedure ExtraitListe(Clef:string;ListSource:Tliste;Var ListBut:Tliste);
{extrait une sous-liste selon une clef principale}
var i:integer;
begin
 if ListSource.Quantite>0 then {la liste n'est pas vide}
 begin
  ListBut.Effacer;
  for i:=1 to ListSource.Quantite do
   if pos(clef,ListSource.Element(i))<>0 then
    ListBut.Ajouter(ListSource.Element(i))
 end
 else
 begin
  Form1.BitBtnCreeListHomme.enabled:=false; {désactive le bouton création liste Hom}
  Form1.BitBtnCreeListFemme.enabled:=false; {désactive le bouton création liste Fem}
 end
end;

procedure TForm1.BitBtnCreeListFemmeClick(Sender: TObject);
{bouton de création de la liste Fem}
begin
 ExtraitListe(KFemme,ListeIndiv,ListeFemme);
 BitBtnvoirListFemme.enabled:=true;
end;

procedure TForm1.BitBtnCreeListHommeClick(Sender: TObject);
```

```
{bouton de création de la liste Hom}
begin
 ExtraitListe(KHomme,ListeIndiv,ListeHomme);
 BitBtnvoirListHomme.enabled:=true;
end;

{-------------------- VISUALISATION DES LISTES -------------------------}

procedure TForm1.BitBtnvoirListFemmeClick(Sender: TObject);
begin
 ListeFemme.VersListBox(ListBox1); {visualise sur écran la liste}
end;

procedure TForm1.BitBtnvoirListHommeClick(Sender: TObject);
begin
 ListeHomme.VersListBox(ListBox1); {visualise sur écran la liste}
end;

procedure TForm1.BitBtnListEntiereClick(Sender: TObject);
begin
 ListeIndiv.VersListBox(ListBox1); {visualise sur écran la liste}
end;

{-------------------------- EFFACEMENT ECRAN --------------------------}

procedure TForm1.BitBtnEffaceEcranClick(Sender: TObject);
begin
 ListBox1.clear;
end;

end.
```

Ici aussi le lecteur est encouragé à modifier et à ajouter de nouvelles fonctionnalités en se servant du source fourni dans l'exemple comme base de travail.

Chapitre 8.2 Les messages Windows

Plan du chapitre:

1. La programmation dirigée par les messages

1.1 Que sont et à quoi servent les messages
1.2 Les messages systèmes
1.3 Delphi et les messages

2. Mécanisme de la répartition des messages en Delphi

2.1 property OnMessage
2.2 procedure MainWndProc
2.3 procedure Dispatch
2.4 Interception directe d'un message
2.5 procedure DefaultHandler
2.6 Gestionnaire d'événement

3. Création et envoi de message en Delphi

3.1 Envoyer un message avec PostMessage et SendMessage
3.2 Exemple d'utilisation

1. La programmation dirigée par les messages

La programmation événementielle peut être présentée comme deux attitudes à adopter lors de la construction d'une application dans un système comme Windows :

> Réagir à des événements (des messages système).
> Engendrer des messages (pour simuler des événements).

Dans les deux cas la notion de messages est présente, ce sont donc des outils de base de ce type de programmation.

1.1 Que sont et à quoi servent les messages

Les messages sont la base de la communication à l'intérieur du système d'exploitation. Ils sont le **résultat d'événements** que le système intercepte ou de **communications du système** avec ses différents composants.

Certaines actions extérieures ou intérieures au système déclenchent des événements. Windows génère alors en continu, un flot de messages en direction des applications ou vers d'autres parties du système lui-même dans le cas de messages internes. Comme son nom l'indique Windows s'intéresse tout particulièrement aux fenêtres, donc pour chaque fenêtre présente, ces messages sont stockés dans une file d'attente (de type FIFO) et sont traités au fur et à mesure par le système.

Certains messages de cette file d'attente sont associés à des événements particuliers. Un tel message peut ainsi être récupéré par une application : nous dirons alors que l'application "**réagit à l'événement**".

Rappellons que si l'on se place soit du point de vue de l'utilisateur, soit de celui d'une application (*ce qui est notre cas*), Windows devient alors semblable à une grande boucle qui attend un événement d'où qu'il vienne, le stocke dans la file des messages, puis le traite. Voici pour une fenêtre le pseudo-comportement de Windows :

```
tantque non ArrêtSysteme faire
  si événement alors
   construire Message ;
   si Message ≠ ArrêtSysteme alors
      reconnaître la fenêtre à qui est destinée ce Message;
      distribuer ce Message
   fsi
  fsi
ftant
```

Dans ce document, la **partie traitement** des messages destinés à un contrôle d'une fenêtre ou d'une application Delphi, est le seul mécanisme qui nous intéresse dans notre programmation événementielle. Nous pouvons utilement et très schématiquement, matérialiser un tel traitement sous la forme de la boucle suivante :

tantque FIFO des messages non vide **faire**
  lire en tête de FIFO le **Message**;
  construire une structure associée à ce **Message**;
  utiliser la méthode interne de traitement sur ce **Message**
ftant

Remarque

> Pendant que ce traitement a lieu, Windows continue en outre à intercepter d'autres événements et à stocker dans la file d'attente les messages associés à ces événements et les messages internes.

En résumé nous pouvons énoncer l'affirmation suivante :

Le système **envoie** ou **poste** des messages prédéfinis à une application, mais réciproquement une application peut **envoyer** ou **poster** des messages vers d'autres fenêtres ou d'autres applications.

1.2 Les messages systèmes dans Windows

Chaque message système dispose d'un identificateur unique du message. Il correspond à une **constante numérique** de base de Windows. Chaque identificateur de message est associé à une action spécifique.

Le classement des messages dans Windows s'effectue par un préfixe qui permet de déterminer la catégorie à laquelle appartient le message.

Soit par exemple le message dont l'identificateur est **WM_KEYDOWN**, il appartient à la catégorie des messages généraux de Windows **WM_**xxx. C'est un message associé à l'appui d'une touche de clavier. Le préfixe de l'identificateur indique le type de fenêtre qui peut intercepter et donc réagir à ce message.

> Les messages **WM_**xxx sont des **W**indow **M**essages donc destinés à toutes les fenêtres.
> Les messages **LB_**xxx sont des **L**ist **B**ox messages destinés aux boîtes de listes.
> Les messages **EM_**xxx sont des **E**dit **M**essages destinés aux contrôles d'édition.
> Les messages **SBM_**xxx sont des **S**croll **B**arre **M**essages destinés aux barres de défilement.
> etc...

L'aide en ligne de l'API W32 de Windows nous fournit à toutes fins utiles, la structure générale d'un message de Windows :

voici cette structure en C :	voici cette structure en Delphi :
typedef struct tagMSG { *// msg* HWND **hwnd**; UINT **message**; WPARAM **wParam**; LPARAM **lParam**; DWORD **time**; POINT **pt**; } MSG;	**type** TMsg = **packed record** **hwnd**: HWND; **message**: UINT; **wParam**: WPARAM; **lParam**: LPARAM; **time**: DWORD; **pt**: TPoint; **end**;

Ci-dessous la signification des champs de la structure d'un message :

hwnd = la référence de la fenêtre à qui est destinée le message.
message = la **constante numérique** identifiant le message.
wParam = entier positif codé sur quatre octets [0..4294967295] représentant un premier paramètre dont la signification dépend de la catégorie du message.
lParam = entier quelconque codé sur quatre octets [–2147483648..2147483647] représentant un deuxième paramètre dont la signification dépend de la catégorie du message.
time = heure système à laquelle le message a été créé par Windows.
pt = position du curseur de souris en coordonnées d'écran (pt.x, pt.y :Longint)

Exemple:

Lors de l'appui sur une touche clavier que va-t-il se passer ?

Si vous tapez sur la touche "M" du clavier, une série de messages est envoyée en cascade :

- Un message **WM_KEYDOWN** est automatiquement envoyé par Windows dans la file d'attente de la fenêtre détenant la focalisation (nommons la **FenWin**), dès que la touche est enfoncée, et ceci tant que la touche est enfoncée.
- Ensuite Windows envoie le message **WM_CHAR** (qui contient le caractère entré au clavier) dans la même file;
- puis lorsque vous relâchez la touche, il envoie enfin un message **WM_KEYUP**.

Que contient par exemple, la structure **TMsg** lors du traitement de ce message **WM_KEYDOWN** *(information obtenue à partir de l'aide en ligne de l'API Win32)*:

hwnd: référence (pointeur,adresse) de la fenêtre **FenWin**.
message: **WM_KEYDOWN**,(valeur numérique=256)
wParam: = 77 (code virtuel de la touche M ici)
lParam: les 32 bits de ce mot indiquent les informations suivantes :

- bits 0..15 : le nombre sur 16 bit indique combien de fois il faut répéter le caractère. Il correspond au maintien de la touche enfoncée.

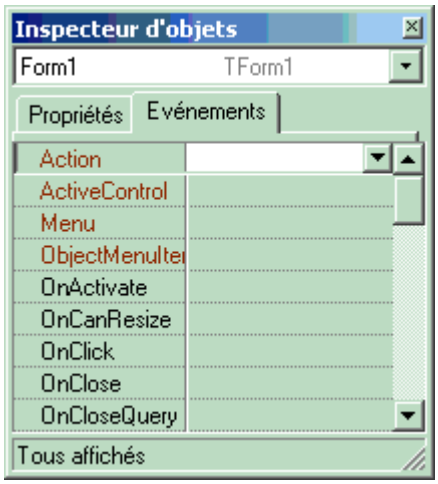
- bits 16..23 : un code spécifique au clavier (OEM)
- bit 24 : si bit = 1 c'est une touche étendue (ALT+,CTRL+,...) sinon bit = 0
- bit 25..28 : réservé par le système.
- bit 29 : bit de contexte = 0 pour ce message.
- bit 30 : bit d'état précédent de cette touche si bit = 1 la touche est déjà enfoncée, sinon la touche était relevée et alors bit = 0
- bit 31 : bit d'état de transition = 0 pour ce message.

1.3 Delphi et les messages

Une application Delphi est basée sur une fenêtre d'application. Nommons notre application **FenDelphi;** elle est donc considérée par Windows comme une fenêtre quelconque et à ce titre elle peut recevoir et envoyer des messages.

Une application Delphi possède des outils d'interception de la structure **TMsg**, de son décodage et de son interprétation. Nous avons déjà vu que Delphi disposait, à l'attention du programmeur, pour chaque objet et pour certains événements, de **gestionnaires d'événement.**

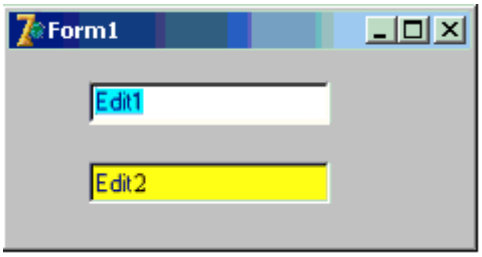
Ces gestionnaires sont des outils de premier niveau (en fait les plus abstraits et les plus encapsulés)de gestion des messages. L'inspecteur d'objet nous a permis de nous familiariser avec de tels gestionnaires dans son onglet événements.



Onglet - événements

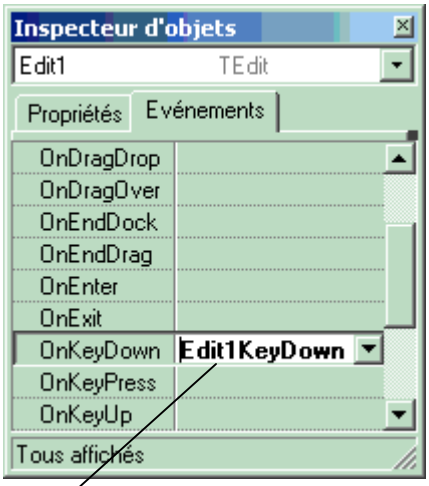
Si nous reprenons l'exemple précédent de l'appui sur la touche "M" du clavier lorsque **FenDelphi** détient la focalisation, nous savons que Windows va construire la structure **TMsg** précédente et l'ajouter dans la file d'attente des messages de **FenDelphi**.

Supposons que dans notre application **FenDelphi** nous ayons déposé sur la fiche **Form1** deux contrôles visuels **Edit1** et **Edit2** de la classe des **TEdit,** et que ce soit **Edit1** qui détienne le focus lors du lancement de l'application **FenDelphi** :



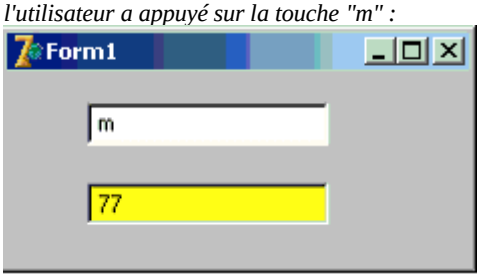
*Edit détient le focus dans Form1 de **FenDelphi***

Nous souhaitons faire réagir **Edit1** à l'appui sur la touche "M" du clavier de la façon suivante : dès que l'utilisateur appuie sur une touche du clavier dans **Edit1**, il apparaît dans **Edit2** le code de ce caractère.
Nous allons utiliser le gestionnaire de l'événement OnKeyDown de **Edit1** :




```
procedureTForm1.Edit1KeyDown(Sender: TObject; var Key: Word;  Shift: TShiftState);
begin
  Edit2.text:=inttostr(Key) //code du caractère entré dans Edit, recopié dans Edit2
end;
```

l'utilisateur a appuyé sur la touche "m" :



Voici l'aide en ligne de DELPHI sur l'événement **OnKeyDown** :

```
type
  TKeyEvent = procedure (Sender: TObject; var Key: Word; Shift: TShiftState) of object;
property OnKeyDown: TKeyEvent;
```

- Le gestionnaire d'événement **OnKeyDown** permet d'effectuer un traitement spécifique quand une touche est enfoncée. Le gestionnaire **OnKeyDown** peut répondre à toutes

les touches du clavier, y compris les touches de fonction et les combinaisons avec les touches Maj, Alt et Ctrl ainsi qu'avec les boutons de la souris.

- Le paramètre Key indique la touche du clavier.
- Le type TKeyEvent pointe sur une méthode (un gestionnaire d'événement) gérant les événements du clavier : ici TForm1.Edit1KeyDown
- **OnKeyDown** est déclenché automatiquement lorsque le message **WM_KEYDOWN** est intercepté par l'application.

Il existe donc un mécanisme interne à Delphi qui a permis à notre application de reconnaître que Windows avait envoyé le message **WM_KEYDOWN** dans la structure **TMsg**, ce mécanisme a permis aussi d'appeler le gestionnaire **TForm1.Edit1KeyDown** que nous avons écrit afin d'effectuer la recopie du code dans Edit2.

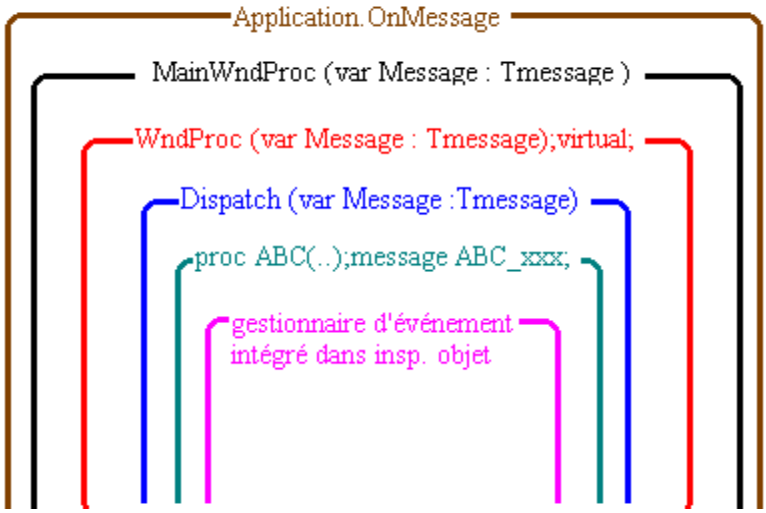
2. Mécanisme de répartition des messages en Delphi

Delphi répartit les messages de plusieurs façons :

Chaque composant hérite d'un système complet de répartition de message. Toutes les classes Delphi disposent d'un mécanisme intégré pour gérer les messages : ce sont les méthodes de gestion des messages ou **gestionnaires de messages,** ces méthodes sont choisies dans un ensemble de méthodes spécifiques.

Le système de répartition de message dispose d'une gestion par défaut. Vous ne définissez de gestionnaire que pour les messages auxquels vous souhaitez spécifiquement répondre. Un gestionnaire par défaut est appelé si aucune méthode n'est définie pour le message.

Le diagramme suivant illustre une partie essentielle du fonctionnement du système de répartition de message dans Delphi et les différents niveaux d'interception possibles d'un message :



property OnMessage: TMessageEvent;
procedure MainWndProc(**var** Message: TMessage);
procedure WndProc(**var** Message: TMessage); **virtual**;

```
procedure Dispatch(var Message); virtual;
procedure ABCxxx(var Message: TMessage);message ABC_xxx;
procedure DefaultHandler(var Message); virtual;
```

Procédons à l'examen de chaque niveau d'interception

2.1 niveau : property OnMessage

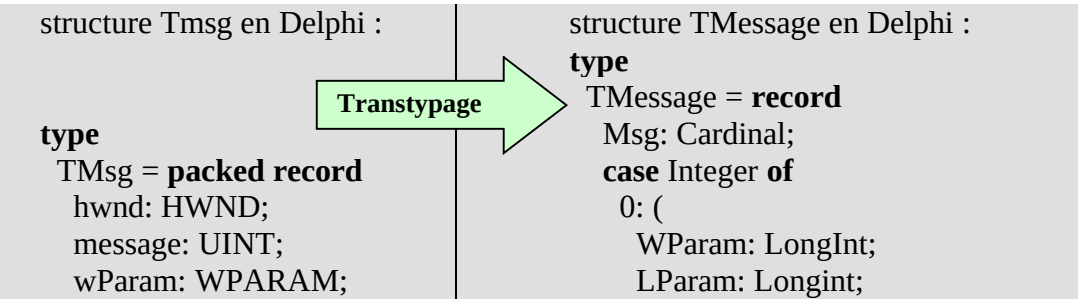
Si vous voulez intercepter tout ou partie des messages Windows expédiés à toutes les fenêtres de l'application Delphi, utilisez l'événement OnMessage de la classe **TApplication** qui encapsule toute application Delphi. Cet événement OnMessage est déclenché dès que votre application reçoit un quelconque message de Windows.

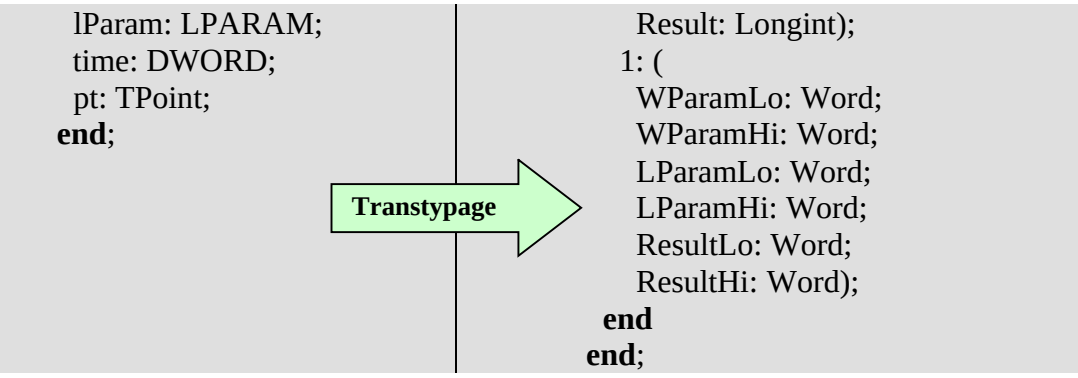
```
property OnMessage: TMessageEvent;
type TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object;
```

L'écriture du gestionnaire d'événement OnMessage vous permet de répondre à tous les messages que reçoit l'application:

```
type
 TForm1 = class(TForm)
   ........
   private
   procedure MonAppliMessages(var Msg: TMsg; var Handled: Boolean);
 end;

implementation

procedure TForm1.MonAppliMessages(var Msg: TMsg; var Handled: Boolean);
begin
 //traitement du message contenu dans Msg du type TMsg
   if Msg.message=WM_LBUTTONDOWN then ....Traitement
end;
//Le gestionnaire étant créé par l'instruction Application.OnMessage:=MonAppliMessages :
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnMessage := MonAppliMessages;
end;
```

si vous ne créez pas un tel gestionnaire le message est distribué à la fenêtre à laquelle il est destiné et Delphi continue la suite de tentative de traitement du message pour la fenêtre concernée. Tout d'abord Delphi transtype la struture TMsg en une structure interne de type TMessage :





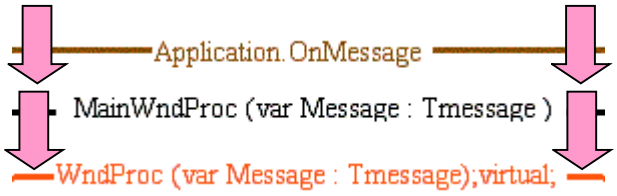
Le champ hwnd n'a plus de raison d'exister puisque la fenêtre à qui le message s'adresse a déjà été identifiée. Les champs time et pt de la structure **TMsg** sont réintégrés à l'intérieur d'autres champs de la structure **TMessage**.

Remarque

> OnMessage **ne reçoit que des messages envoyés à la file d'attente Fifo des messages**, et non ceux envoyés directement (en particulier ceux de SendMessage).

2.2 niveau : procedure MainWndProc - WndProc

L'interception ayant eu lieu la redirection vers la fenêtre adéquate est assurée par la **procedure** MainWndProc(**var** Message: TMessage); c'est la **procedure** WndProc(**var** Message: TMessage); **virtual**; surchargeable, qui va se charger de traiter ou transmettre le message.



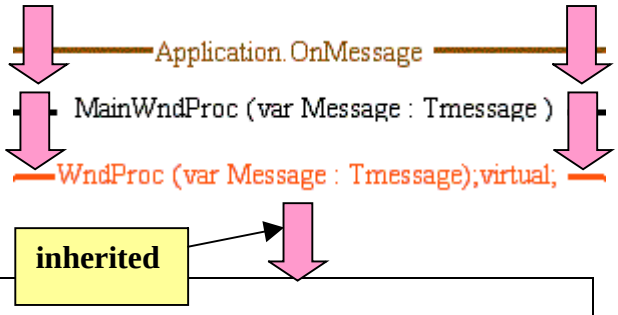
Exemple :
soit dans une application à intercepter l'enfoncement du bouton gauche de la souris.

```
type
 TForm1 = class(TForm)
   ........
   private
   procedure WndProc (var Message: TMessage); override;
 end;

implementation

procedure TForm1.WndProc(var Message: TMessage);
begin
   if Message.Msg = WM_LBUTTONDOWN then ....Traitement
end;
```

Le traitement ci-dessus de WM_LBUTTONDOWN dans WndProc arrête la diffusion des messages vers les autres niveaux d'encapsulation.

Le code de la procédure WndProc tel qu'il est écrit ci-dessus est inefficace et peut produire un message d'erreur car de tous les messages passant par WndProc nous n'avons décidé de n'en traiter qu'un seul le : WM_LBUTTONDOWN, en oubliant les autres messages de la fenêtre comme le positionnement, l'affichage etc...

Pour laisser Delphi s'occuper du reste il suffit de rajouter **inherited** à la fin de WndProc et la transmission des autres messages pourra se poursuivre.

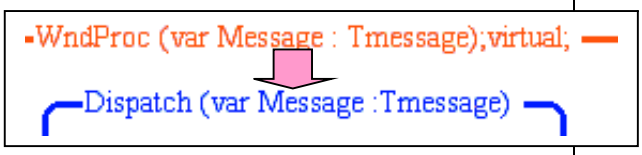



```
type
 TForm1 = class(TForm)
   ........
   private
   procedure WndProc(var Message: TMessage); override;
 end;

implementation

procedure TForm1.WndProc(var Message: TMessage);
begin
  if Message.Msg = WM_LBUTTONDOWN then ....Traitement
  inherited;
end;
```

2.3 niveau : procedure Dispatch

La méthode WndProc reçoit tous les messages destinés à la fenêtre et fait un filtrage. Elle appelle la méthode Dispatch pour chaque message selectionné, cette méthode n'a pas de type au message qui est transmis, c'est un message générique. Si vous voulez intercepter un message spécifique à ce niveau vous devrez le transtyper par exemple en TMessage.

```
type
 TForm1 = class(TForm)
   ........
   private
   procedure Dispatch(var Message); override;
 end;

implementation

procedure TForm1.Dispatch(var Message);
begin
```



```
  if  Tmessage (Message).Msg = WM_LBUTTONDOWN then ....Traitement
end;
```

2.4 niveau : Interception de redéfinition d'un message

procedure ABCxxx(**var** Message: TMessage);**message** ABC_xxx;

Ce mécanisme direct vous autorise à créer une méthode spécifique en redéfinissant le traitement d'un message donné du système Windows. Il s'agit d'une redéfinition, car le message ABC_xxx est traité par la classe, la **procedure** ABCxxx(**var** Message: TMessage) remplace ce traitement habituel (liaison statique) par le vôtre. L'utilisation d'inherited se justifie pleinement si l'on souhaite laisser le traitement habituel avoir lieu.

En reprenant le même exemple d'enfoncement du bouton gauche de souris :

```
type
 TForm1 = class(TForm)
   ...........
   private
     procedure WMLBUTTONDOWN(var Message:Tmessage);message WM_LBUTTONDOWN;
 end;

implementation

procedure TForm1.WMLBUTTONDOWN(var Message:Tmessage);
begin
  ....Traitement
 {si vous mettez inherited et qu'un gestionnaire de l'événement a été écrit par le  programmeur pour
un événement déclenché par WM_LBUTTONDOWN, vous pouvez laisser le message continuer à être
traité.}
end;
```

Au chapitre précédent dans le composant TwinArbre nous construit une telle procédure :

```
private
 procedure WMSize(var Message:TWMsize); message WM_SIZE;
 ......
procedure TwinArbre.WMSize(var Message:TWMsize);
{permet de modifier la taille du TCustomcontrol lors de la conception}
begin
   inherited;
   Ftree.setbounds(0,0,width,height-25);
   FPotentiometre.setbounds(0,FTree.Top+FTree.Height,width-25,25);
   FEdit1.setbounds(FPotentiometre.left+FPotentiometre.width, Ftree.Top+Ftree.Height,25,25);
end;
```

Elle intercepte et redéfini le message WM_SIZE, qui est envoyé au composant TCustomControl dès qu'une de ses dimensions est modifiée. Nous avons mis inherited afin de laisser se propager le message WM_SIZE aux niveaux en-dessous et afin que le TCustomControl réagisse correctement , puis nous avons programmé le redimensionnement de chacun des composants visuels déposé sur le TcustomControl.

2.5 niveau : procedure DefaultHandler

Si aucun gestionnaire n'a été écrit pour ce message, le système de gestion par défaut de Delphi s'en charge à votre place : il appelle la méthode **DefaultHandler**. Comme pour Dispatch le paramètre est de type message générique. Il s'agit ici d'un mécanisme qui se trouve juste au dessus de celui du gestionnaire d'événement.
DefaultHandler permet en particulier de gérer par surcharge tous les messages pour lesquels l'objet n'a pas de gestionnaire spécifique ou bien d'intercepter (c'est le denier niveau possible) un message connu par l'objet.

```
type
 TForm1 = class(TForm)
   ...........
   private
     procedure DefaultHandler(var Message); override;
 end;

implementation

procedure TForm1.DefaultHandler(var Message);
begin
  if  TMessage(Message).Msg = WM_LBUTTONDOWN then ....Traitement
end;
```

2.6 niveau : Gestionnaire d'événement

Enfin si un gestionnaire d'événement a été écrit par le programmeur, la méthode Dispatch l'appelle. Il s'agit des gestionnaires d'événements qui sont fournis par Delphi avec une entrée dans l'inspecteur d'objet, ici pour l'interception de l'enfoncement de la souris l'événement OnMouseDown est géré. L'événement **OnMouseDown** est déclenché automatiquement lorsque l'un des messages suivants **WM_LBUTTONDOWN**, **WM_MBUTTONDOWN**, **WM_RBUTTONDOWN** est intercepté par l'application :

```
type
 TForm1 = class(TForm)
   procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
     Shift: TShiftState; X, Y: Integer);
   private
     ........
 end;

implementation

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
     Shift: TShiftState; X, Y: Integer);
begin
  ....Traitement
end;
```

*Exemple : affichage de la hiérarchie des niveaux d'interception du **WM_LBUTTONDOWN***

Dans cet exemple nous interceptons le click gauche de souris à tous les niveaux, et nous laissons l'interception continuer aux niveaux inférieurs grâce au **inherited**. Ci-dessous le code source de la unit du programme Delphi correspondant :

```
Unit Unitexemple;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Memo1: TMemo;
    Label1: TLabel;
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Déclarations privées }
    procedure MonAppliMessages(var Msg: TMsg; var Handled: Boolean);
    procedure WndProc(var Message: TMessage); override;
    procedure Dispatch(var Message);override;
    procedure DefaultHandler(var Message); override;
    procedure WMLBUTTONDOWN(var Message:Tmessage);message WM_LBUTTONDOWN;
  public
    { Déclarations publiques }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
 Application.OnMessage:= MonAppliMessages;
end;
procedure TForm1.MonAppliMessages(var Msg: TMsg; var Handled: Boolean);
begin
  if Msg.message=WM_LBUTTONDOWN then
  begin
   Edit1.text:=inttostr(Msg.wParam);
   Edit2.text:=inttostr(Msg.lParam);
   Edit3.text:=inttostr(Msg.time);
   Edit3.color:=clBlue;
   Memo1.Lines.Add('intercepté : niveau 1 - MonAppliMessages')
  end;
  inherited
end;
procedure TForm1.WndProc(var Message: TMessage);
begin
 if Message.Msg = WM_LBUTTONDOWN then
 begin
  Edit1.text:=inttostr(Message.WParam);
  Edit2.text:=inttostr(Message.LParamHi) + '/' + inttostr(Message.LParamLo);
  Edit3.text:=inttostr(Message.Result);
  Edit3.color:=clred;
```

```
  Memo1.Lines.Add('intercepté : niveau 1 - WndProc')
 end;
 inherited;
end;
procedure TForm1.Dispatch(var Message);
begin
 if  Tmessage(Message).msg = WM_LBUTTONDOWN then
 begin
  Edit1.text:=inttostr(Tmessage(Message).WParam);
  Edit2.text:=inttostr(Tmessage(Message).LParamHi) + '/' + inttostr(Tmessage(Message).LParamLo);
  Edit3.text:=inttostr(Tmessage(Message).Result);
  Edit3.color:=clyellow;
  Memo1.Lines.Add('intercepté : niveau 2 - Dispatch')
 end;
 inherited;
end;
procedure TForm1.WMLBUTTONDOWN(var Message:Tmessage);
begin
  Edit1.text:=inttostr(Message.WParam);
  Edit2.text:=inttostr(Message.LParamHi) + '/' + inttostr(Message.LParamLo);
  Edit3.text:=inttostr(Message.Result);
  Edit3.color:=cllime;
  Memo1.Lines.Add('intercepté : niveau 3 - Proc WMLBut') ;
  inherited;
end;
procedure TForm1.DefaultHandler(var Message);
begin
if Tmessage(Message).msg = WM_LBUTTONDOWN then
 begin
  Edit1.text:=inttostr(Tmessage(Message).WParam);
  Edit2.text:=inttostr(Tmessage(Message).LParamHi)+'/'+inttostr(Tmessage(Message).LParamLo);
  Edit3.text:=inttostr(Tmessage(Message).Result);
  Edit3.color:=clAqua;
  Memo1.Lines.Add('intercepté : niveau 4 - DefaultHandler')
 end;
 inherited;
end;
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 Edit3.color:=cllime;
 Memo1.Lines.Add('intercepté : niveau 5 - gest. OnMouseDown') ;
end;
end.
```

Si l'on click sur le fond de la fiche voici l'ordre d'interception en cascade du message WM_LBUTTONDOWN :

Remarquons que l'Edit3 est en couleur clLime, ce qui est normal puisque c'est le niveau 6 du gestionnaire d'événement OnMouseDown qui est exécuté en dernier.

Si l'on click dans n'importe lequel des contrôles déposés sur la fiche (un des Edit, ou le memo) voici ce que nous obtenons :

C'est le gestionnaire MonAppliMessages de l'événement OnMessage de l'application qui a été le seul à être exécuté. Ce qui signifie que l'application a bien reçu de la part du système le message WM_LBUTTONDOWN et que son gestionnaire MonAppliMessages l'a pris en compte. Toutefois comme nous avions clické dans l'Edit2 (Edit jaune) pour la suite Delphi n'ayant aucun traitement proposé par le programmeur, a abandonné le traitement de ce message.

3. Création et envoi de message en Delphi

Il est possible de travailler avec des messages définis par l'utilisateur. Une application Delphi traitera ces messages comme elle traite les autres messages système.

Vous devez déclarer un identificateur de message personnel. Un identificateur de message est une constante entière numérique. Windows se réserve pour son propre usage les messages dont le numéro est inférieur à 1024. Lorsque vous déclarez vos propres messages, vous devez donc toujours débuter par un numéro supérieur ou égal à 1024.

Delphi vous fournit si vous souhaitez l'utiliser, un identificateur de constante **WM_USER** représentant le numéro de départ (WM_USER = 1024) pour vos messages.

Voici trois constantes de messages personnels :

```
const
 MonMessage1 = WM_USER + 100;
 MonMessage2 = WM_USER + 120;
 MonMessage3 = 1500;
```

3.1 Envoyer un message avec PostMessage et SendMessage

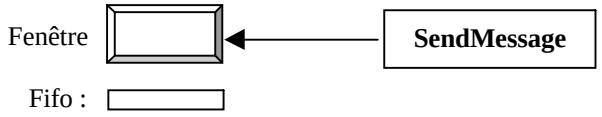
L'API Windows met à notre disposition de nombreuses procédures de bas niveau permettant d'envoyer un message à un contrôle fenêtré (descendant des TWinControl), dont les plus simples sont **PostMessage** et **SendMessage**. Il est rare que nous soyons obligés d'utiliser ce genre de fonction de bas niveau, mais ici l'objectif est d'indiquer comment envoyer des messages personnalisés.

SendMessage (procédure de traitement synchrone) attend que le message soit traité, si le message est envoyé à un autre thread, l'application émettrice du message reste bloquée tant que l'application receptrice n'a pas fini de le traiter.
en-tête de la fonction :

function SendMessage(hwnd : HWND; Msg : cardinal; wParam , lParam : Longint) : Longint;

Remarque

SendMessage envoie directement le message à une fenêtre **sans passer par la file d'attente**. L'événement OnMessage de l'application n'est donc pas déclenché, il est conseillé de n'utiliser **SendMessage** qu'à l'intérieur d'une même application pour envoyer des messages synchrones à des contrôles fenêtrés de l'application (Fiche, Panel, Edit, etc...). Dans le cas où vous utilisez **SendMessage pour communiquer avec une autre application,** pensez bien à construire dans l'application réceptrice, une procédure d'interception directe du message personnalisé, ce qui sera le seul moyen d'intercepter ce message.

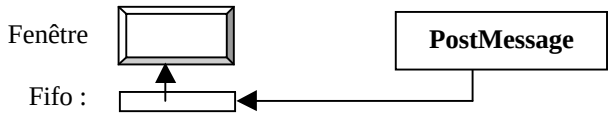


PostMessage (procédure de traitement asynchrone)envoie le message dans la file d'attente des messages du contrôle fenêtré et n'attend pas que le message soit traité.
en-tête de la fonction :

function PostMessage(hwnd : HWND; Msg : cardinal; wParam , lParam : Longint) : Longint;

Remarque

PostMessage envoie le message à une procédure de fenêtre dans **la file d'attente de la fenêtre**. L'événement OnMessage de l'application est déclenché. Dans le cas où vous utilisez **PostMessage pour communiquer avec une autre application** vous pouvez intercepter le message personnalisé comme avec SendMessage au niveau procédure d'interception directe, mais aussi au niveau de L'événement OnMessage de l'application.



Bien entendu, outre vos messages personnalisés, ces deux procédures permettent aussi d'envoyer des messages définis de Windows (WM_xxx, LB_xxx, EM_xxx, etc...)

SendMessage(Form1.Handle,WM_CLOSE,0,0); *//message de fermeture de la fiche Form1*
SendMessage(Form1.Edit1.Handle,WM_CHAR,ord('M'),0); *//envoi du caractère 'M' dans Edit1*

etc...

Ces deux fonctions de bas niveau, ont comme premier paramètre le Handle de fenêtre du contrôle à qui est envoyé le message, il existe pour les contrôles en général, une méthode notée **Perform** qui permet d'envoyer directement un message à la procédure de fenêtre du contrôle sans avoir à connaître son Handle : le contrôle répond alors comme s'il avait reçu le message Windows spécifié.

function Perform(Msg: Cardinal; WParam, LParam: Longint): Longint;

Exemple d'utilisation

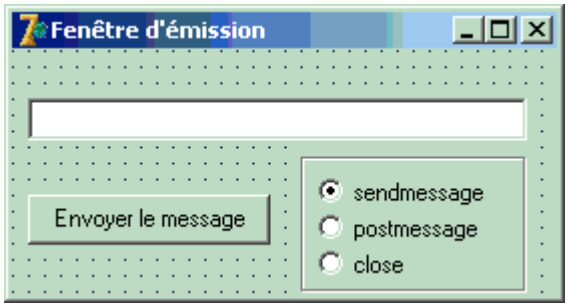
Enoncé :
Afficher le même message WM_USER expédié soit par PostMessage, soit par SendMessage, soit fermer la fenêtre principale d'une application PReception.exe, *à partir d'une autre application* PEmissPost.exe, *par d'envoi de messages.*

Application émettrice : **PEmissPost**

Application réceptrice : **Preception**

1 - Source de l'application PEmissPost.exe

Cette application envoie le message WM_USER (si l'un des deux radio-bouton sendmessage ou postmessage est coché), si c'est le radio bouton close qui est coché elle envoie le message WM_USER+1 :



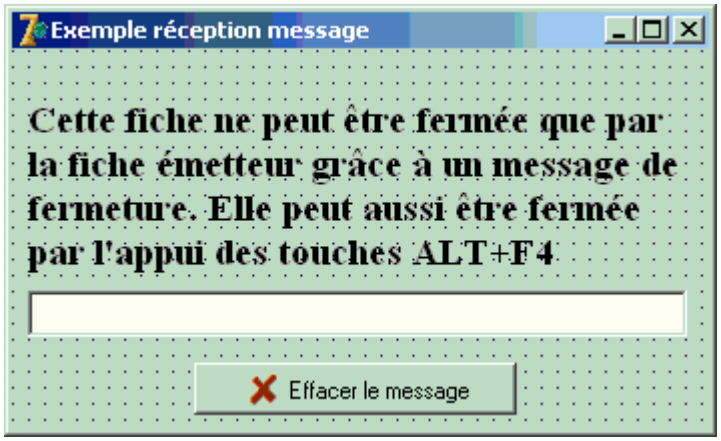
code source complet de l'application

```
unit UFemissPost;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

 type
  TForm1 = class(TForm)
  Button1: TButton;
  EditInfos: TEdit;
  RadioGroupTypeMessage: TRadioGroup;
  procedure Button1Click(Sender: TObject);
  procedure RadioGroupTypeMessageClick(Sender: TObject);
 private
```

```
{ Déclarations privées }
public
{ Déclarations publiques }
 procedure EnvoyerUnMessage;
end;

 var  Form1: TForm1;

implementation

{$R *.dfm}
{------------ Emission  message fermeture à la fenêtre TFExemple ----(émission)---------}
// fermeture de la fenêtre TFxxxx déjà ouverte avant fermeture
procedure TForm1.EnvoyerUnMessage;
var
 Wnd:HWnd;
 begin
  Wnd:=FindWindow('TFExemple',nil); // recherche de la fenêtre TFExemple
  if Wnd<>0 then  // si instance de la fenêtre  trouvée => on lui envoi un message
  begin
   EditInfos.Text:='Fenêtre TFExemple trouvée, message envoyé !';
   case
    radiogrouptypemessage.itemindex of
    0: SendMessage(Wnd,WM_USER,0,0); // message à traiter directement
    1: PostMessage(Wnd,WM_USER,0,0); // message posté dans la fifo
    2: PostMessage(Wnd,WM_USER+1,0,0); // message close posté dans la fifo
   end;
  end
  else
   EditInfos.Text:='Fenêtre TFExemple non trouvée';
 end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 EnvoyerUnMessage;
end;

procedure TForm1.RadioGroupTypeMessageClick(Sender: TObject);
begin
 EditInfos.Text:=''
end;

end.
```

2 - Source de l'application PReception.exe

Cette application reçoit des messages en particulier ceux provenant de l'application précédente lorsque les deux applications sont exécutées en même temps, nous avons programmé la reception et l'interception des messages WM_USER et WM_USER+1 à trois niveaux : 1°)niveau OnMessage, 2°) niveau WndProc , 3°) niveau redéfinition de message.

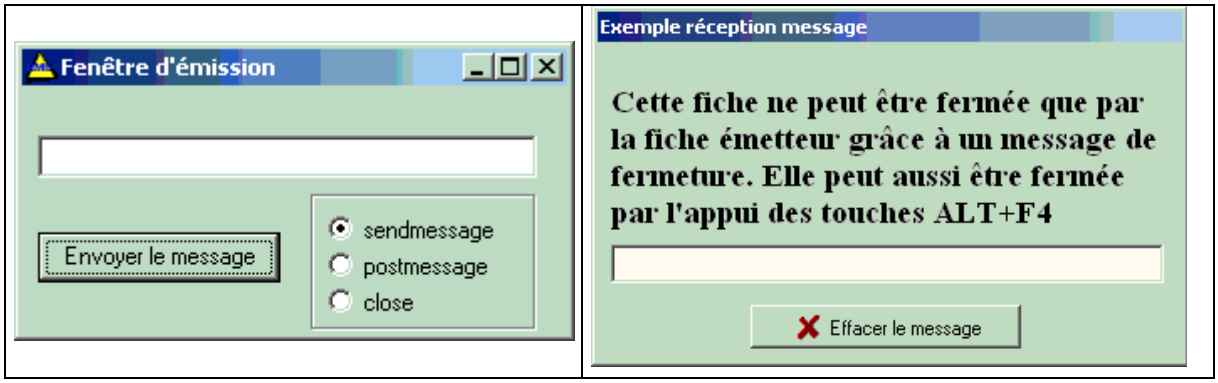


code source complet de l'application

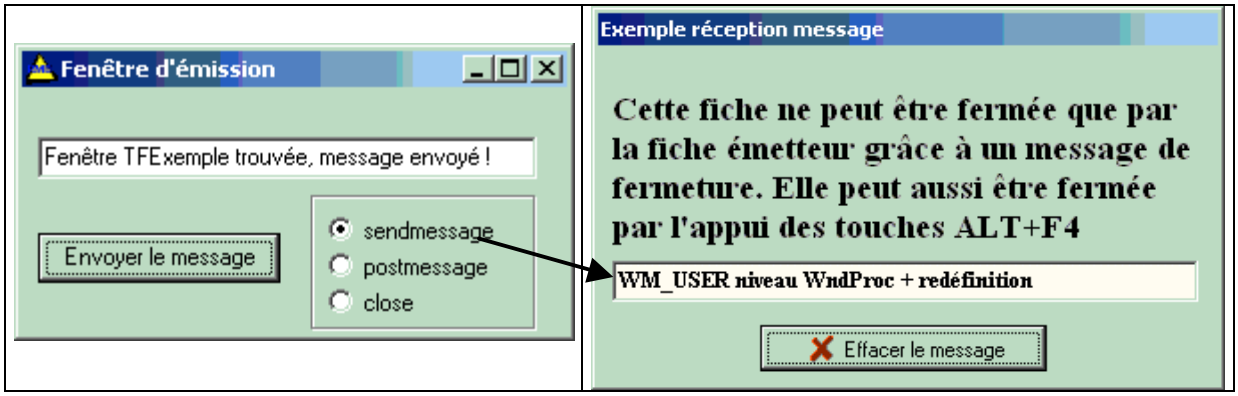
```
unit UFReception;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, Buttons;

 type
  TFExemple = class(TForm)
  Label1: TLabel;
  Edit1: TEdit;
  BitBtnEffacer: TBitBtn;
  procedure FormCreate(Sender: TObject);
  procedure BitBtnEffacerClick(Sender: TObject);
 private
 { interception à 3 niveaux différents}
  procedure MessageUser(var mess:TMsg;var hand:boolean);
  procedure WndProc(var Message: TMessage);override;
  procedure MessWMUSER (var Mess:TMessage);message WM_USER;
 public
end;

 var
  FExemple: TFExemple;

implementation

 {$R *.dfm}
 {---------------- Interception message utilisateur ----(reception)---------------}
 procedure TFExemple.MessageUser(var mess:TMsg;var hand:boolean);
 { traite le message wm_User intercepté comme un ordre donné à la fenêtre.
 Méthode personnelle de traitement des messages reçus de la part d'une autre application }
 begin
  if mess.message=WM_USER then // message envoyé = WM_USER
  begin
   Edit1.Text:='WM_User OnMessage + ';
  end
  else
   if mess.message=WM_USER+1 then // message envoyé = WM_USER+1 (<=>close ici)
   begin
     close  // message reçu interprété ici comme une fermeture "close"
   end;
  {else if mess.message=WM_USER+2 then .....}
```

```
 inherited;
end;

procedure TFExemple.WndProc(var Message: TMessage);
begin
 if Message.msg=WM_USER then // message envoyé = WM_USER (<=>close ici)
 begin
  if Edit1.Text='' then
     Edit1.Text:='WM_USER ';
  Edit1.Text:=Edit1.Text+'niveau WndProc';
 end;
 inherited
end;

procedure TFExemple.MessWMUSER (var Mess:TMessage);
begin
 Edit1.Text:=Edit1.Text+' + redéfinition';
end;

procedure TFExemple.FormCreate(Sender: TObject);
begin
 Application.OnMessage:=MessageUser; //autorise l'interception des messages expédiés à cette fenêtre
end;

procedure TFExemple.BitBtnEffacerClick(Sender: TObject);
begin
 Edit1.Text:=''
end;

end.
```

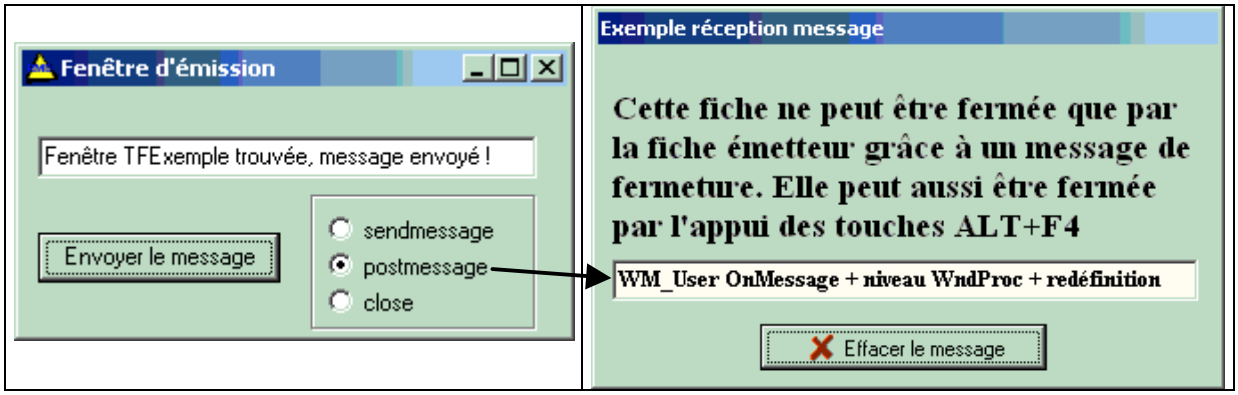
On lance les deux applications **PReception.exe** et **PEmissPost.exe** , voici l'état initial des deux fenêtres de chaque application lancée en même temps :



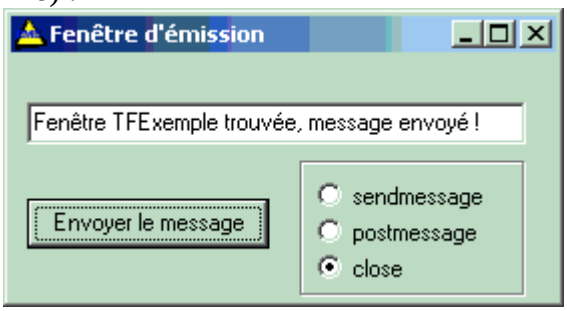
En cliquant sur le bouton"Envoyer le message" comme le radio-bouton sendmessage est sélectionné, c'est l'instruction **"SendMessage(Wnd,WM_USER,0,0);"**qui est exécutée (rappelons qu'elle ne met rien dans la fifo de la fenêtre et que donc OnMessage ne l'intercepte pas, ce que l'on vérifie dans les images d'exécution ci-après :



Si l'on sélectionne le radio-bouton postmessage et que l'on clique sur le bouton"Envoyer le message", c'est l'instruction "**PostMessage(Wnd,WM_USER,0,0);**"qui est exécutée (rappelons qu'elle met le message dans la fifo de la fenêtre et que donc OnMessage l'intercepte, ce que l'on vérifie dans les images d'exécution ci-après :



Enfin la fiche de droite se ferme lors du click sur le bouton "Envoyer le message" lorsque le radio-bouton close est sélectionné (car c'est le message WM_USER+1 qui est posté dans la fifo) :



Il est conseillé d'exécuter cet exemple sur sa machine, puis de le modifier en testant d'autres niveaux d'interception, en bloquant le traitement par suppression du mot inherited afin de bien se familiariser avec l'utilisation pratique du mécanisme de répartition des messages avec Delphi.

Chapitre 8.3 Créer un événement associé à un message

Plan du chapitre:

1. Rappel sur la construction d'un événement

2. Exemple de création d'événements dans un TEdit

2.1 Evènement OnColorChange
2.2 Evènement OnResize

1. Rappel sur la construction d'un événement

Nous avons déjà fourni une notice méthodologique de construction d'un événement au chapitre 5.5, rappelons rapidement en le complétant ce qu'il faut savoir pour élaborer un nouvel événement :

Voici les étapes qui interviennent dans la définition d'un événement :

- Définition du type de gestionnaire
- Déclaration de l'événement
- Appel de l'événement

Pour construire un nouvel événement dans ClasseA :

- Il nous faut d'abord définir un type pour l'événement : EventTruc
- Il faut ensuite mettre dans ClasseA une propriété d'événement : **property** OnTruc : EventTruc
- Il faut créer un champ privé nommé FOnTruc de type EventTruc en lecture et écriture qui servira de champ de stockage de la propriété OnTruc.
- Si l'on pense à la réutilisation de la classe par héritage : Il est utile de créer une méthode centralisatrice **protected** virtuelle d'appel du pointeur de méthode FOnTruc, que les développeurs construisant une nouvelle classe héritant de ClasseA redéfiniront dans leur class fille.

Précédemment nous avons vu comment définir deux événements sur une structure de données de pile Lifo, nous rappelons ci-dessous une partie du code que nous allons modifier :

```
type
  DelegateLifo = procedure ( Sender: TObject ;  s :string ) of  object ;

 ClassLifo = class (TList)
 private
    FOnEmpiler : DelegateLifo ;
    FOnDepiler : DelegateLifo ;
 public
   function  Est_Vide  : boolean ;
   procedure  Empiler (elt : string ) ;
   procedure  Depiler ( var  elt : string ) ;
   property  OnEmpiler : DelegateLifo  read  FOnEmpiler  write  FOnEmpiler ;
   property  OnDepiler : DelegateLifo  read  FOnDepiler  write  FOnDepiler ;
 protected
   procedure  EvtEmpiler (elt : string ) ; virtual;
   procedure  EvtDepiler (elt : string ) ; virtual;
end;
```

Définition du type de gestionnaire

Déclaration de l'événement

Méthodes centralisatrices pour l'appel de l'événement

Les appels au pointeur de méthode (qui pointe vers le futur gestionnaire de l'événement) se situent à des endroits stratégiques choisis pour représenter la survenue de l'événement choisi et ont lieu à travers la méthode virtuelle protected :

```
Implementation
procedure  EvtEmpiler (s : string ) ;
begin
    if  assigned(FOnEmpiler)  then
        FOnEmpiler ( self , s )
end;

procedure  EvtDepiler (s : string ) ;
begin
    if  assigned(FDeEmpiler)  then
        FOnDepiler ( self , s )
end;

procedure  ClassLifo.Depiler( var  elt : string ) ;
 begin
  if  not  Est_Vide  then
  begin
    elt :=string (self.First) ;
    self.Delete(0) ;
    self.Pack ;
    self.Capacity := self.Count ;
    EvtDepiler (elt)
   end
  end;

 procedure  ClassLifo.Empiler(elt : string ) ;
 begin
   self.Insert (0 , PChar(elt)) ;
   EvtEmpiler (elt)
 end;
```

Appel de l'événement

Appel de l'événement

Ceci représente la construction générale de tout événement, ce qui peut différer d'une construction à l'autre c'est la façon et le lieu de l'appel de l'événement. Nous allons voir comment utiliser les messages système pour créer des événements en particulier dans les TControl.

2. Création de deux nouveaux événements dans un TEdit

Mise en oeuvre sur l'adjonction de 2 évènements nouveaux à un composant déjà existant.

Enoncé : construire dans une classe dérivant des TEdit

- un événement OnColorChange qui permet de notifier à l'utilisateur un changement dans la couleur d'un TEdit et de programmer éventuellement la gestion de cet événement.
- un événement OnResize qui permet de signaler que le TEdit vient de voir changer sa taille (l'une de ses deux propriétés width ou height).

2.1 Evénement OnColorChange

2.1.1 Déclenchement de l'événement
2.1.2 Définition du gestionnaire
2.1.3 Déclaration de l'événement
2.1.4 Propriété d'accès à l'événement
2.1.5 Appel de l'événement par procédure
2.1.6 Le gestionnaire vide de l'événement

2.1.1 Déclenchement de l'événement

Comme nous ne sommes pas maître du code source de base des TEdit, il ne nous est pas possible de travailler comme dans l'exemple précédent de la pile Lifo. Il nous faut essayer de trouver s'il existe un message système adéquat et alors l'intercepter pour l'utiliser à nos fins.

Après observation de la classe des TEdit, nous n'avons pas trouvé de message système informant le TEdit que sa couleur vient de changer : soit nous devons abandonner ,soit nous cherchons un message qui se rapporte à l'action qui a lieu lorsque la couleur vient de changer. Nous remarquons que dans l' éventualité du changement de couleur du fond d'un contrôle, le système redessine le contrôle.

Nous optons donc pour la redéfinition l'interception du message système WM_PAINT qui est envoyé très regulièrement à la fenêtre d'un contrôle afin que celui-ci se redessine. Nous avons vu au chapitre précédent comment déclarer la méthode d'interception du dit message, nous décidons de nommer cette méthode WMPaintChgColor :

```
procedure WMPaintChgColor (var Msg : TWMpaint); message WM_PAINT;
```

Le type TWMPaint est défini par Delphi pour coder après transtypage un message système WM_PAINT

```
TWMPaint = packed record
  Msg: Cardinal;
  DC: HDC;
  Unused: Longint;
  Result: Longint;
end;
```

2.1.2 Définition du gestionnaire

Nous construisons un gestionnaire d'événements spécifiques, car nous souhaitons disposer en paramètre de la nouvelle couleur.

```
TEventColor = procedure (Sender: TObject; couleur:Tcolor) of object;
```

2.1.3 Déclaration de l'événement

```
private     FColorChange : TEventColor;
```

2.1.4 Propriété d'accès à l'événement

```
published
   property OnColorChange : TEventColor read FColorChange write FColorChange;
```

2.1.5 Appel de l'événement par une procédure centralisatrice

```
protected
   procedure ColorChange(coul:TColor);virtual;
```

2.1.6 Le gestionnaire vide de l'événement OnColorChange doit être valide

```
if Assigned(FColorChange) then
   FColorChange(self,coul)
```

2.2 Evénement OnResize

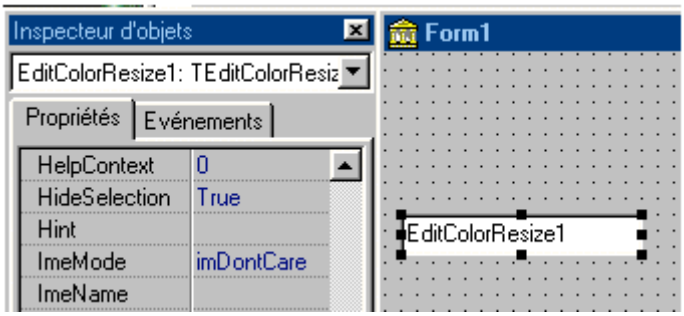
L'événement OnResize a été construit de la même manière que l'événement OnColorChange, par interception du message WM_SIZE, toutefois à titre d'exemple il n'y a pas de procédure surchargeable centralisatrice du genre **procedure** ColorChange. Dans ce cas le programmeur ne pourra pas rajouter un comportement spécifique sur l'événement OnResize. Nous ne répètons pas la démarche qui est strictement identique.

Ci-dessous, nous donnons le code source du composant simple dérivé des TEdit :

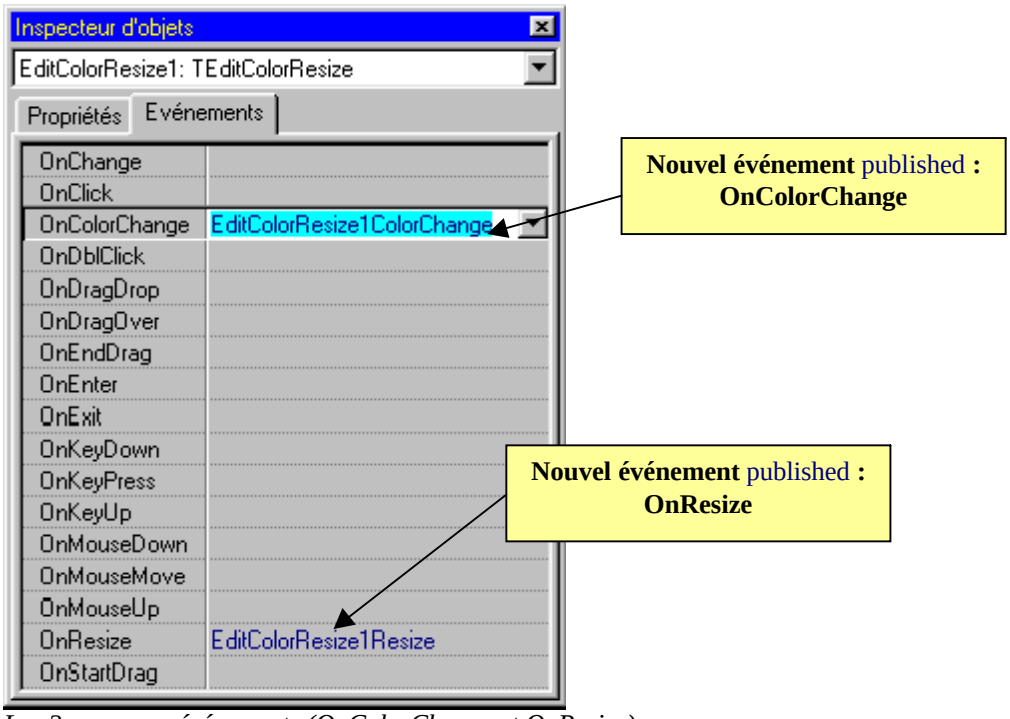
```
unit UEditColor;
interface

 uses
  SysUtils, Messages, Classes, Graphics, Controls,  Forms, StdCtrls, ExtCtrls, Outline;
type
  TEventColor = procedure (Sender: TObject; couleur:Tcolor) of object;
  { pointeur de gestionnaire d'événement OnColorChange (paramètre couleur : TColor, nécessaire ici }
  TEditColorResize = class(TEdit)
```

```
      private
       FColorChange : TEventColor;
      { pointeur de méthode spécifique permettant utiliser la property OnColorChange }
       ColorPrec:Tcolor; // la couleur précédente
      FOnResize:TNotifyEvent;
     { pointeur de méthode permettant utiliser la property OnResize }
       procedure WMPaintChgColor(var Msg:TWMpaint); message WM_PAINT;
     {le message WM_PAINT qui permettra de matérialiser l'événement}
      procedure WMSizeRedim(var Msg:TWMsize); message WM_size;
     {le message WM_size est envoyé au TEdit dès son height ou son width a changé }
     protected
        procedure ColorChange(coul:TColor);virtual;   {pour surcharge ultérieure par le programmeur }
     public
        constructor Create(AOwner: TComponent); override;
     published  //nouveau gestionnaire d'événement
        property OnColorChange : TEventColor read FColorChange write FColorChange;
        property OnResize:TNotifyEvent read FOnResize writeFOnResize;
     end;
   procedure Register;

   implementation

   procedure Register;
   begin
     RegisterComponents('Exemples', [TEditColorResize]);
   end;
   //-------------------------------------------------------------//
   constructor TEditColorResize.Create(AOwner: TComponent);
   begin
    inherited;
    color:=clwhite;
    ColorPrec:=self.color // initialisation à la couleur actuelle
   end;
   procedure TEditColorResize.WMSizeRedim(var Msg:TWMsize);
   // intercepte le message WM_size et traite le changement de taille
   begin
    if Assigned(OnResize) then
      FOnResize(self) //lien avec le futur gestionnaire du programmeur
   end;
   procedure TEditColorResize.WMPaintChgColor(var Msg:TWMpaint);
   // intercepte le message WM_Paint et traite le changement de couleur
   begin
    inherited ;
    if color<ColorPrec then
     begin
      ColorChange(ColorPrec);
      ColorPrec:=color;
      change // appelle le gestionnaire d'événement OnChange, s'il est défini.
     end
   end;
   procedure TEditColorResize.ColorChange(coul:TColor);
   // pour que le développeur puisse la surcharger éventuellement
   begin
     if Assigned(FColorChange) then
      FColorChange(self,coul) // lien avec le futur gestionnaire du programmeur
   end;
end.
```

Enregistrez ce composant dans votre Delphi, puis testez-le sur les deux événements nouveaux en écrivant un projet de test et en programmant les deux gestionnaires d'événements.
Voici après dépôt d'un TEditColorResize ce que l'inspecteur d'objet nous montre sur cette classe:



toutes les propriétés sont celles des TEdit



Les 2 nouveaux événements (OnColorChange et OnResize)

Nous montrons ci-dessous comment un développeur utilisant notre composant TEditColorResize peut surcharger et implanter un comportement additionnel à l'événement *OnColorChange*.

```
interface
TEditUser = class(TEditColorResize)
  protected
   procedure ColorChange(coul:TColor);override;
end;
implementation
procedure TEditUser.ColorChange(coul:TColor);
begin
 inherited; // appel à la procédure centralisée de la classe ancêtre
 //...comportement nouveau rajouté par le programmeur
end;
```

Chapitre 8.4 ActiveX avec la technologie COM

Plan du chapitre:

Introduction

1. Les notions d'interfaces COM de microsoft

La notion d'interface en général avec Delphi
1.1 Qu'est-ce qu'une interface COM
1.2 L'interface Iunknown
1.3 Les CoClasses en Delphi
1.4 l'architecture COM et le registre Windows

2. ActiveX est un objet COM

3. Création d'un ActiveX avec Delphi

1°) Construction à partir du composant
2°) Afficher l'expert contrôle ActiveX
3°) Recenser l'ActiveX
4°) Comment installer dans la palette Delphi un ActiveX

4.Déploiement et utilisation Web d'une fiche ActiveX

1°) Construction de la fiche de base
2°) Recensement de la fiche ActiveX
3°) Déploiement sur le web

Introduction

COM signifie **C**omponent **O**bject **M**odel, c'est le modèle proposé par Microsoft pour créer des composants logiciels distribués. La principale fonction de COM que nous voyons dans ce chapitre consiste à construire des composants fenêtrés nommés ActiveX qui peuvent être utilisés sur Internet à travers "Internet Explorer" et développés et manipulés par des langages différents. Les ActiveX peuvent être développés comme les composants Delphi, dans un environnement de développement Delphi, C++, VB6,… et être réutilisés dans l'un quelconque de ces environnements. Les ActiveX peuvent aussi être convertis en Winforms dans la plate-forme .**Net** et être ainsi réutilisés par les langages de .**Net** comme : C#, VB Net,…

La famille de systèmes d'exploitation Windows utilise cette architecture, et de très nombreux développements ont été centrés sur cette norme. L'architecture .**Net** de microsoft se positionne comme remplaçant de COM tout en gardant dores et déjà une compatibilité ascendante avec celle-ci.

1. Les notions d'interfaces COM de microsoft

La notion d'interface en général avec Delphi

- Une interface est un contrat, elle peut contenir des **propriétés**, des **méthodes** mais **ne** doit contenir **aucun champ** ou **attribut**.
- Une interface **ne** peut **pas** contenir des méthodes déjà implémentées.
- Tous les membres d'une interface sont **publics** sans nécessiter de qualificateur de visibilité.
- Une interface est héritable.
- Pour pouvoir construire un objet à partir d'une interface, il faut définir une classe non abstraite implémentant **toutes** les méthodes de l'interface.
- En Delphi une interface se déclare avec le mot clef **interface**, aux mêmes endroits qu'une déclaration de classe.

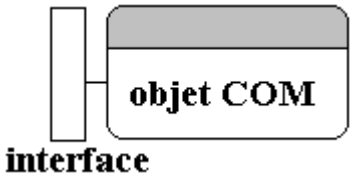
Exemple de déclaration d'interface en Delphi :

Animal = **Interface**
  procedure SeDeplacer;
  procedure Manger;
  function NombreDePattes : integer;
end;
	Indique des méthodes de fonctionnement d'un animal :
	❑ Comment se déplace-t-il
	❑ Comment mange-t-il
	❑ Combien a-t-il de pattes,

Oiseau = **Interface** (Animal) **procedure** ConstruireNid; **end;**	Un oiseau possédera les 3 méthodes précédentes + une méthode expliquant comment il construit son nid.

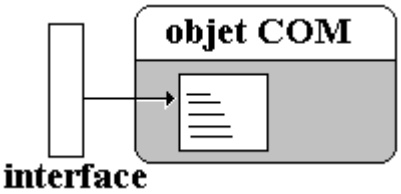
1.1 Qu'est-ce qu'une interface COM

Les clients COM communiquent avec des objets par le biais d'interfaces COM. Les interfaces sont des groupes de routines, qui assurent la communication entre le fournisseur d'un service (objet serveur) et ses objets clients.

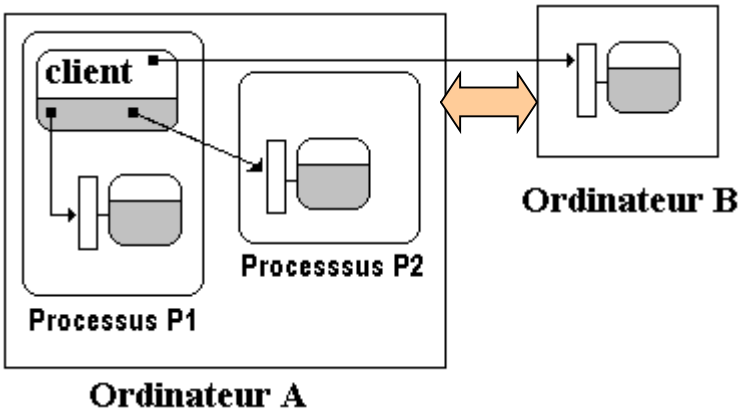


Rappelons qu'une interface représente juste un *contrat*, c'est-à-dire que la classe chargée d'implémenter cette interface s'engage à implémenter **toutes** les méthodes déclarées dans l'interface. **L'interface est l'unique moyen** de mettre à disposition du client le service fourni par l'objet, sans lui donner les détails de l'implémentation sur la façon dont ce service est fourni.

Une interface COM est **un pointeur dans un objet COM**, elle pointe en fait sur la table des méthodes de l'objet COM. Une interface n'est donc pas un objet, elle est en outre **identifiée de manière unique** par un nombre appelé GUID.



Avec COM, un client n'a pas besoin de savoir où réside l'objet serveur qu'il invoque (que l'objet réside dans le même processus P1 que le client, dans un autre processus P2 sur la machine A du client ou sur une autre machine B du réseau). C'est COM qui met en place les moyens d'accéder à l'objet serveur.



1.2 L'interface IUnknown

Nous avons précisé plus haut qu'afin d'obtenir des objets indépendants (du langage, de la plate-forme d'exécution,...) nous devions pouvoir disposer d'une interface de base commune ou universelle.

L'interface **IUnknown** est l'interface de base commune et universelle du **modèle COM.** Elle doit être présente dans tout langage implantant l'architecture COM.

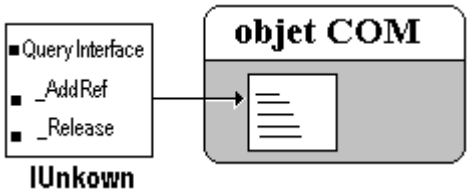
En Delphi toute interface hérite directement ou indirectement de **Iunknown.** (c'est d'ailleurs ce qui rend la notion d'interface plus lourde qu'en Java ou C#)

En Delphi voici la déclaration de **IUnknown** qui possède seulement 3 méthodes :

```
type
IInterface = interface
['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface (const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;

IUnknown = IInterface;
```

QueryInterface	permet de naviguer entre les différentes interfaces implémentées par un objet COM (comme toute interface dispose de cette méthode, on peut accéder à n'importe qu'elle interface à partir de IUnknown).
_AddRef	permet l'incrémentation d'un compteur de références : : lorsqu'un objet se connecte à l'interface, le compteur est incrémenté de 1.
_Release	permet la décrémentation du compteur de référence : lorsqu'un objet se déconnecte de l'interface, le compteur est décrémenté de 1. Ainsi, lorsque le compteur est arrivé à 0, l'objet COM sait qu'il peut libérer la mémoire qu'il occupait.

Chaque langage de programmation (C++, Delphi,...) se charge ensuite d'implanter l'interface **IUnknown** afin d'instancier un objet COM qui dérive obligatoirement et au minimum cette interface **IUnknown**.



En Delphi, c'est la classe **TInterfacedObject** qui est chargée d'implémenter l'interface **IUnknown**. Elle est déclarée comme suit :

```
TInterfacedObject = class(TObject, IInterface)
 protected
   FRefCount: Integer;
   function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
   function _AddRef: Integer; stdcall;
   function _Release: Integer; stdcall;
 public
   procedure AfterConstruction; override;
   procedure BeforeDestruction; override;
   class function NewInstance: TObject; override;
   property RefCount: Integer read FRefCount;
end;
```

1.3 Les CoClasses en Delphi

Dans Delphi un objet COM est nécessairement comme tout objet de Delphi, une instanciation d'une classe Delphi. Pour un objet COM cette classe doit obligatoirement être une classe très spéciale : une **coclasse**.

Un objet COM est en Delphi une instance d'une CoClasse, qui est une classe implémentant une ou plusieurs interfaces COM. L'objet COM fournit les services définis par les méthodes de ses interfaces.

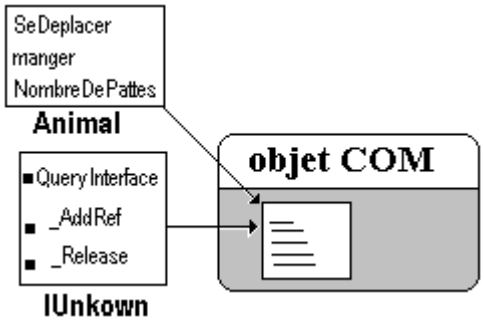
> Une coclasse est une classe chargée d'implémenter une ou plusieurs interfaces. C'est elle qui va "contenir" le code machine et donc constituer le moule de fabrication de l'objet COM.
>
> Une coclasse peut bien sûr dériver d'une classe existante.
>
> Une coclasse doit dériver d'une classe implantant l'interface **IUnknown**.

Voici un exemple de déclaration d'une coclasse :

```
Animal = Interface // hérite automatiquement de IUnknown
    procedure SeDeplacer ;
    procedure Manger;
    function NombreDePattes : integer;
end;

oiseau = class ( Tobject, Animal )
  public
     procedure SeDeplacer ;
     procedure Manger;
     function NombreDePattes : integer;
   private
     function DureeCouvaison : integer ;
end;
```

Il faut implémenter les 6 méthodes : SeDeplacer, Manger, NombreDePattes, QueryInterface, _AddRef , _Release + la nouvelle méthode DureeCouvaison.

Ci-dessous un objet COM de type oiseau :



Toute coclasse dérivant de **TInterFacedObject** hérite de l'implémentation de **IUnknown**. Il est donc conseillé de construire les coclasses permettant d'instancier des objets COM à partir de cette classe.

```
 Animal = Interface // hérite automatiquement de IUnknown
     procedure SeDeplacer ;
     procedure Manger;
     function NombreDePattes : integer;
end;

oiseau = class (TInterFacedObject, Animal )
  public
      procedure SeDeplacer ;
      procedure Manger;
      function NombreDePattes : integer;
   private
     function DureeCouvaison : integer ;
end;
```

Il suffit alors d'implémenter les 3 méthodes : SeDeplacer, Manger, NombreDePattes + la nouvelle méthode DureeCouvaison.

1.4 l'architecture COM et le registre Windows

Pour assurer l'unicité des identificateurs d'objets COM, on utilise des GUIDs (**Globally Unique IDentifiers**), qui comme leur nom l'indique permettent d'avoir des identifiants (presque) uniques au monde.

Ces GUIDs sont codés sur 128 bits et sont souvent représentés sous une forme standard en hexadécimal afin de les rendre plus "lisibles" à l'homme :

Exemple de GUID

{ 27E14E2B-1104-44B2-AE49-9A8778A130EA}

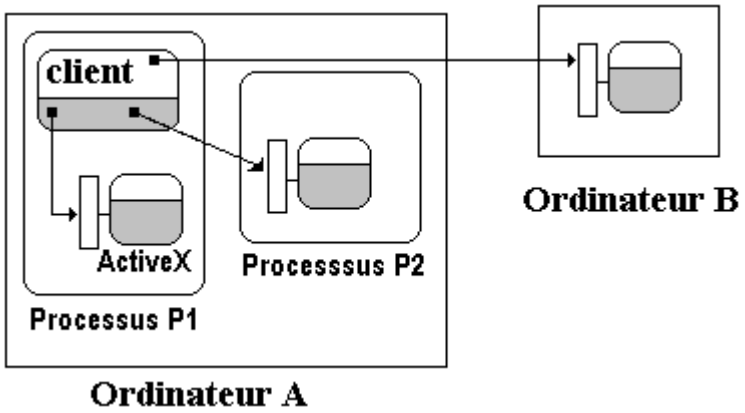
Un GUID peut servir à identifier un objet COM (on parle alors de **CLSID** ou **cl**ass **ID**), une interface (IID), ou bien tout élément que l'on doit identifier très précisément (par exemple une instance d'application).Tout objet COM, pour pouvoir être instancié, doit être au préalable inscrit dans le registre système de Windows (grâce à un utilitaire comme regsvr32.exe du système windows ou tregsvr, qui est livré avec Delphi).

2.ActiveX est un objet COM

Un contrôle ActiveX est un composant logiciel COM qui est inclus dans une application hôte capable de gérer les contrôles ActiveX. Un contrôle ActiveX "étend" les fonctionnalités de l'application conteneur. un tel composant ne nécessite que l'interface **IUnknown**.

Les contrôles **ActiveX** sont conçus pour être eux-mêmes incorporés dans une application client, **ils s'exécutent alors dans le même espace processus** que le client.

Ci-dessous l'application client fait partie du processus P1 ainsi que l'ActiveX qu'elle utilise (les autres objets COM auxquels elle accède ne sont pas des ActiveX). C'est exactement ce qui se passe lorsque l'on "exécute" un ActiveX dans le navigateur "Internet Explorer" :



Un contrôle **ActiveX** en Delphi doit au minimum implanter certaines interfaces nécessaires, dont voici la liste fournit par Borland :

IUnknown, IDispatch, IPersistStreamInit, IOleInPlaceActiveObject, IPersistStorage, IViewObject, IOleObject, IViewObject2, IOleControl, IPerPropertyBrowsing, IOleInPlaceObject, ISpecifyPropertyPages .

Une fois conçu, il pourra être utilisé comme n'importe quel autre composant classique de la VCL de Delphi pour être incorporé à l'application.

La palette des composants de Delphi fournit en standard quatre contrôles ActiveX (certains ont été écrits en C++, d'autre en VB6):

Les paragraphes suivant traitent plus spécialement de méthodes pratiques pour construire des ActiveX en Delphi et le déploiement sur le web pour l'utilisation internet d'unActiveX.

3 Création d'un ActiveX avec Delphi

Delphi fournit depuis la version 4, **heureusement** un **expert de construction des ActiveX** qui implantent incluent et génèrent **automatiquement** le code source associé aux diverses interfaces nécessaires. Le seul travail restant à effectuer est de programmer les nouvelles fonctionnalités du futur ActiveX.

Un contrôle ActiveX dans Delphi est simplement un contrôle VCL encapsulé dans une enveloppe de classe ActiveX.

La démarche à suivre est très simple, l'expert effectuant automatiquement toutes les opérations il suffit de suivre et de répondre aux questions de celui-ci, le seul vrai travail consiste à développer un composant Delphi!

Utiliser l'expert contrôle ActiveX

Pour créer un nouveau contrôle ActiveX à partir d'un contrôle VCL personnalisé, Delphi propose deux experts permettant de construire :

> *Des contrôles ActiveX qui encapsulent des classes VCL.*
> *Des fiches ActiveX semblables aux contrôles, mais dirigées vers le déploiement Web.*

Dans les deux cas l'expert place en fait une enveloppe de classe ActiveX autour d'un contrôle VCL ou d'une fiche et construit le contrôle ActiveX qui contient l'objet.

Construisons pas à pas , en 4 étapes, un ActiveX

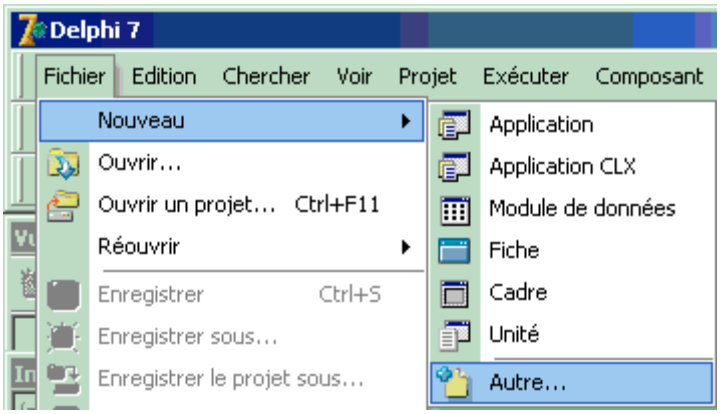
1°) Construction à partir du composant TEditColorResize du sous-chapitre précédent :

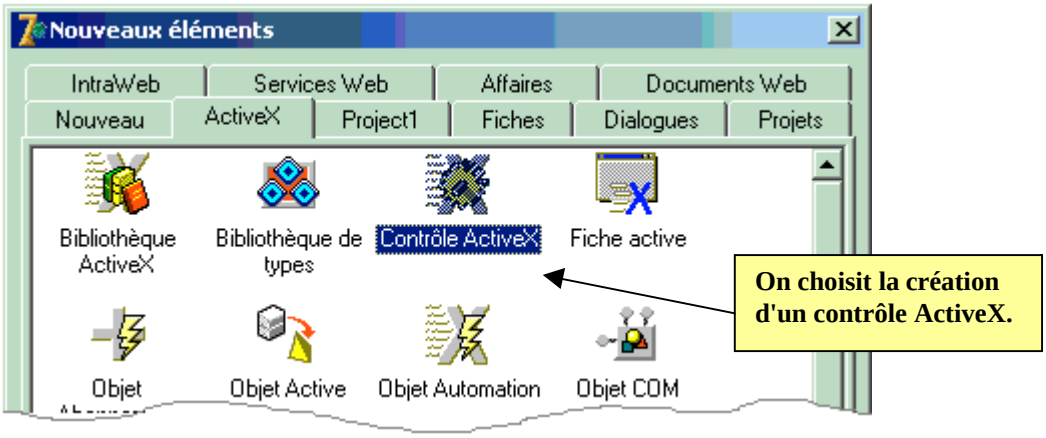
```
unit UEditColor;
interface

 uses
  SysUtils, Messages, Classes, Graphics, Controls,  Forms, StdCtrls, ExtCtrls, Outline;
type
  TEventColor = procedure (Sender: TObject; couleur:Tcolor) of object;
  { pointeur de gestionnaire d'événement OnColorChange (paramètre couleur : TColor, nécessaire ici }
  TEditColorResize = class(TEdit)
   private
    FColorChange : TEventColor;
   { pointeur de méthode spécifique permettant d'utiliser la property OnColorChange }
    ColorPrec:Tcolor; // la couleur précédente
   FOnResize:TNotifyEvent;
  { pointeur de méthode permettant utiliser la property OnResize }
    procedure WMPaintChgColor(var Msg:TWMpaint); message WM_PAINT;
  {le message WM_PAINT qui permettra de matérialiser l'événement}
   procedure WMSizeRedim(var Msg:TWMsize); message WM_size;
  {le message WM_size est envoyé au TEdit dès son height ou son width a changé }
  protected
      procedure ColorChange(coul:TColor);virtual;   {pour surcharge ultérieure par le programmeur }
  public
      constructor Create(AOwner: TComponent); override;
  published  //nouveau gestionnaire d'événement
      property OnColorChange : TEventColor read FColorChange write FColorChange;
      property OnResize:TNotifyEvent read FOnResize writeFOnResize;
  end;
procedure Register;

implementation
```

```
    procedure Register;
    begin
      RegisterComponents('Exemples', [TEditColorResize]);
    end;
    //-------------------------------------------------------------//
    constructor TEditColorResize.Create(AOwner: TComponent);
    begin
     inherited;
     color:=clwhite;
     ColorPrec:=self.color // initialisation à la couleur actuelle
    end;
    procedure TEditColorResize.WMSizeRedim(var Msg:TWMsize);
    // intercepte le message WM_size et traite le changement de taille
    begin
     if Assigned(OnResize) then
       FOnResize(self) //lien avec le futur gestionnaire du programmeur
    end;
    procedure TEditColorResize.WMPaintChgColor(var Msg:TWMpaint);
    // intercepte le message WM_Paint et traite le changement de couleur
    begin
     inherited ;
     if color<ColorPrec then
      begin
       ColorChange(ColorPrec);
       ColorPrec:=color;
       change // appelle le gestionnaire d'événement OnChange, s'il est défini.
      end
    end;
    procedure TEditColorResize.ColorChange(coul:TColor);
    // pour que le développeur puisse la surcharger éventuellement
    begin
      if Assigned(FColorChange) then
       FColorChange(self,coul) // lien avec le futur gestionnaire du programmeur
    end;
end.
```

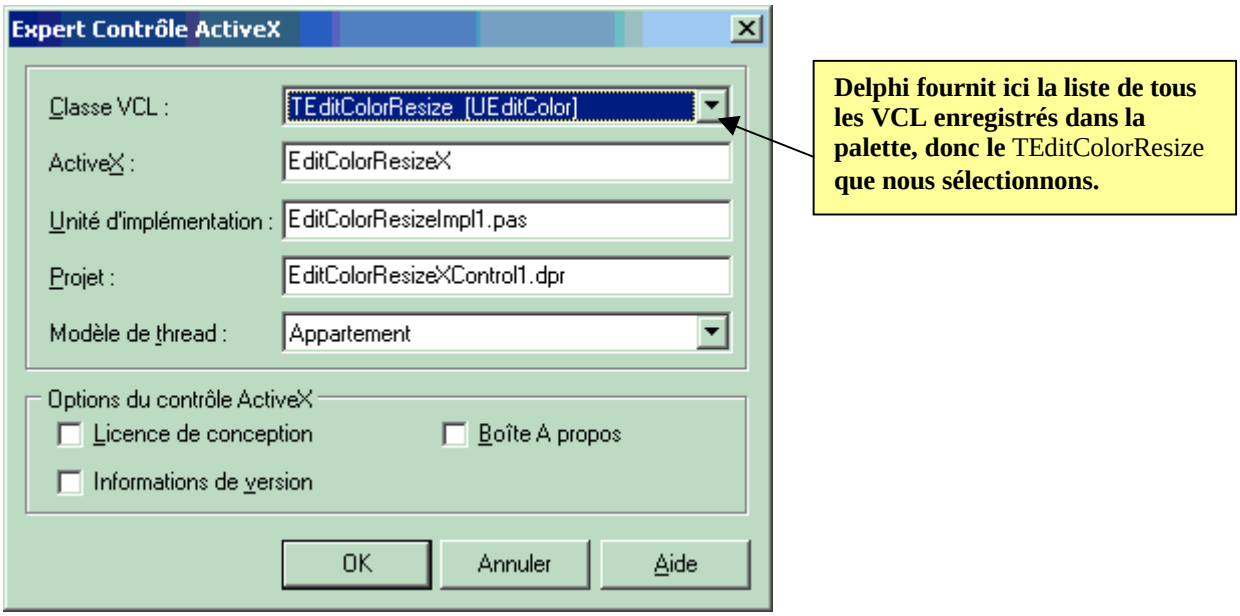
Composant que nous avons rangé dans la palette des composants dans l'onglet exemples :	

2°) Afficher l'expert contrôle ActiveX (à partir du menu Fichier\Nouveau\Autre) :





Delphi envoie un formulaire de saisie d'information sur le paramétrage de création du contrôle ActiveX, nous lui indiquons que nous voulons encapsuler le composant TEditColorResize :



L'expert génère automatiquement des fichiers intermédiaires dont :

Une bibliothèque ActiveX contenant le code nécessaire au démarrage d'un contrôle ActiveX (21 lignes)

```
library EditColorResizeXControl1;

 uses
  ComServ,
  EditColorResizeXControl1_TLB in 'EditColorResizeXControl1_TLB.pas',
  EditColorResizeImpl1 in 'EditColorResizeImpl1.pas' {EditColorResizeX: CoClass};
   {$E ocx}
  exports
            DllGetClassObject,
            DllCanUnloadNow,
            DllRegisterServer,
            DllUnregisterServer;
  {$R *.TLB}
  {$R *.RES}

  begin
  end.
```

Une unit d'ActiveX descendant de TActiveXControl encapsulant le TEditColorResize (560 lignes)

```
unit EditColorResizeImpl1;

 {$WARN SYMBOL_PLATFORM OFF}

 interface

 uses
  Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms, StdCtrls,
  ComServ, StdVCL, AXCtrls, EditColorResizeXControl1_TLB, UEditColor;

 type
  TEditColorResizeX = class(TActiveXControl, IEditColorResizeX)
 private
 { Déclarations privées }
 FDelphiControl: TEditColorResize;
 FEvents: IEditColorResizeXEvents;
  procedure ChangeEvent(Sender: TObject);
  procedure ClickEvent(Sender: TObject);
  procedure ColorChangeEvent(Sender: TObject; couleur: TColor);
  procedure DblClickEvent(Sender: TObject);
  procedure KeyPressEvent(Sender: TObject; var Key: Char);
  procedure ResizeEvent(Sender: TObject);
 protected
 { Déclarations protégées }
  procedure DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage); override;
  procedure EventSinkChanged(const EventSink: IUnknown); override;
  procedure InitializeControl; override;
  function DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
  function Get_AlignDisabled: WordBool; safecall;
  function Get_AutoSelect: WordBool; safecall;
  function Get_AutoSize: WordBool; safecall;
  function Get_BevelInner: TxBevelCut; safecall;
  function Get_BevelKind: TxBevelKind; safecall;
  .........
end;

implementation
 .........
end.
```

L'expert génère aussi : une bibliothèque de type (EditColorResizeXControl1_TLB.pas) qui définit une CoClasse pour le contrôle (540 lignes).

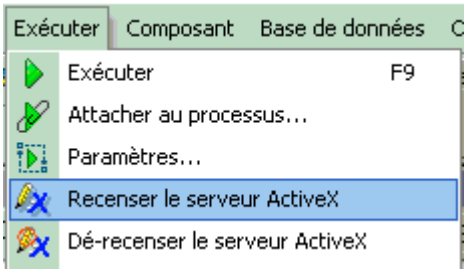
Si nous décomptons les lignes que l'expert a engendré pour encapsuler notre composant, nous obtenons un total de 1120 lignes ! Cela montre malgré tout la lourdeur du codage COM, heureusement nous n'avons pas à intervenir sur ces lignes de codes. Il nous reste à rendre accessible notre nouvel ActiveX afin que d'autres applications puissent l'utiliser.

3°) Recenser l'ActiveX nouvellement créé

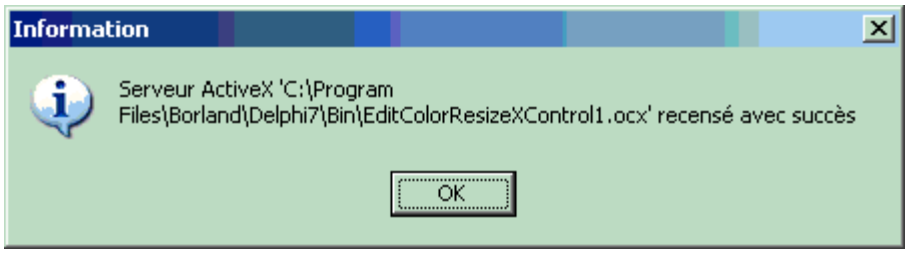
Une fois le contrôle ActiveX créé, il faut le recenser afin que d'autres applications puissent le trouver et l'utiliser c'est à dire l'inscrire dans la base des registres de windows comme serveur

COM. On dispose soit du **regsvr32.exe …** du système d'exploitation (en fenêtre de commande DOS), soit de la commande recenser dans le menuExécuter de Delphi :

Menu Exécuter/Recenser le serveur ActiveX :



Delphi répond :

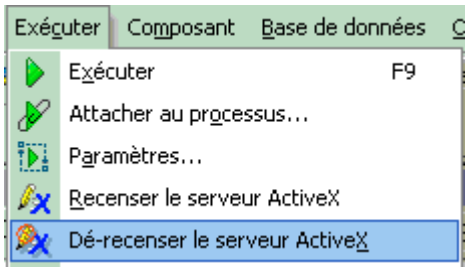


Au final Delphi engendre un seul fichier utile : le fichier contenant l'ActiveX. Les fichiers ActiveX se terminent par le suffixe OCX.

Dans le cas de notre exemple le fichier engendré a pour nom : **EditColorResizeXControl1.ocx.**

Vous pouvez mettre ce fichier où vous le voulez, pour pouvoir l'utiliser il faut le recenser là où vous l'avez mis

Remarque :
Avant de supprimer un contrôle ActiveX du système, il faut annuler son recensement avec le menu *Exécuter|Dérecenser le serveur ActiveX*.



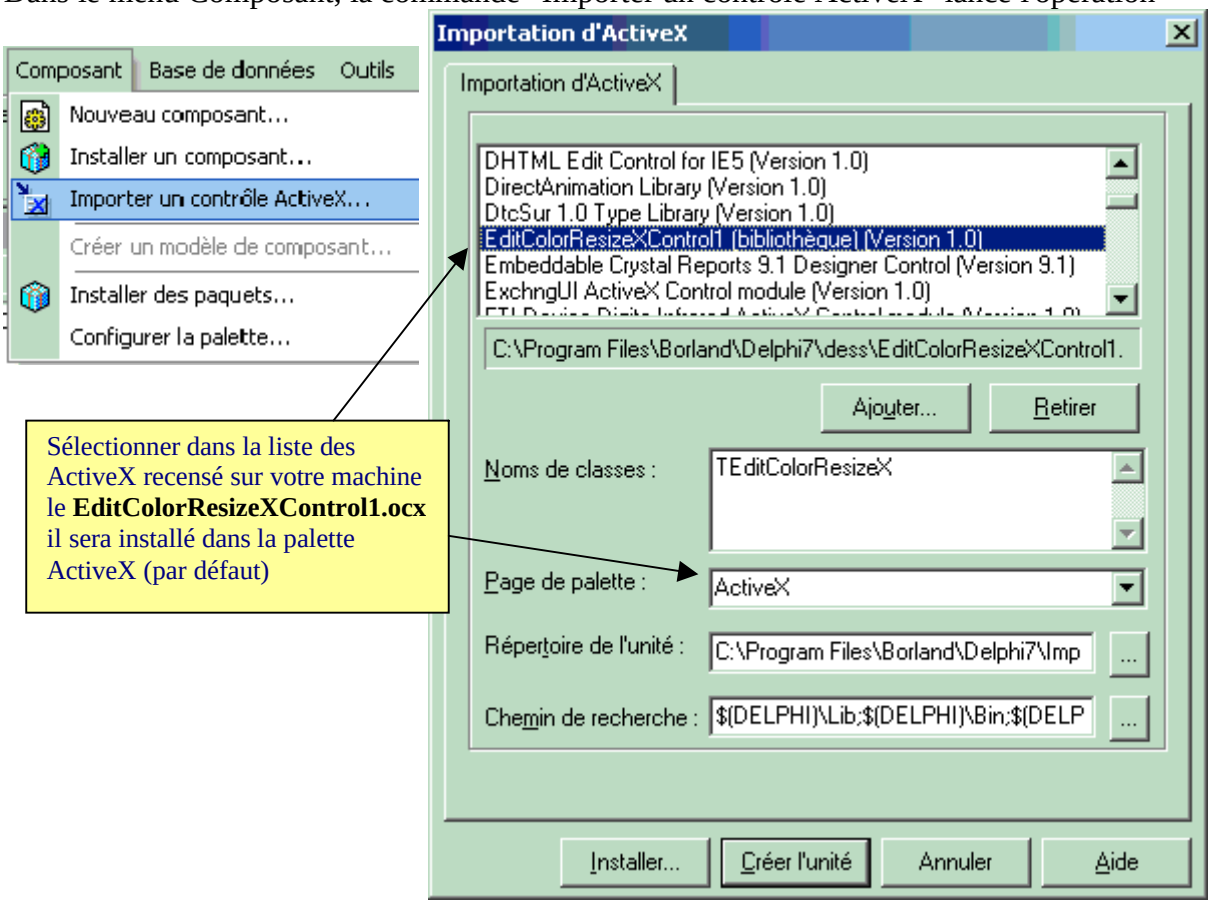
Ou bien lancer la commande **regsvr32.exe /u …** du système d'exploitation (en fenêtre de commande DOS)

Le composant **EditColorResizeXControl1.ocx** est maintenant utilisable avec un autre environnement C++, visual Basic,…

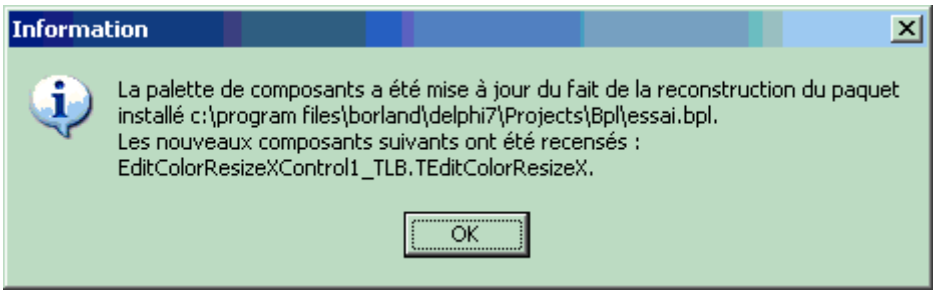
Afin de vérifier le bon fonctionnement de **EditColorResizeXControl1.ocx** nous allons l'installer dans la palette des composants Delphi comme un nouvel ActiveX (Delphi ignore qu'il a été conçu par Delphi, c'est un ActiveX comme des milliers d'autres que vous pouvez acheter ou récupérer gratuitement sur Internet).

4°) Comment installer dans la palette Delphi un ActiveX

Dans le menu Composant, la commande "Importer un contrôle ActiveX" lance l'opération



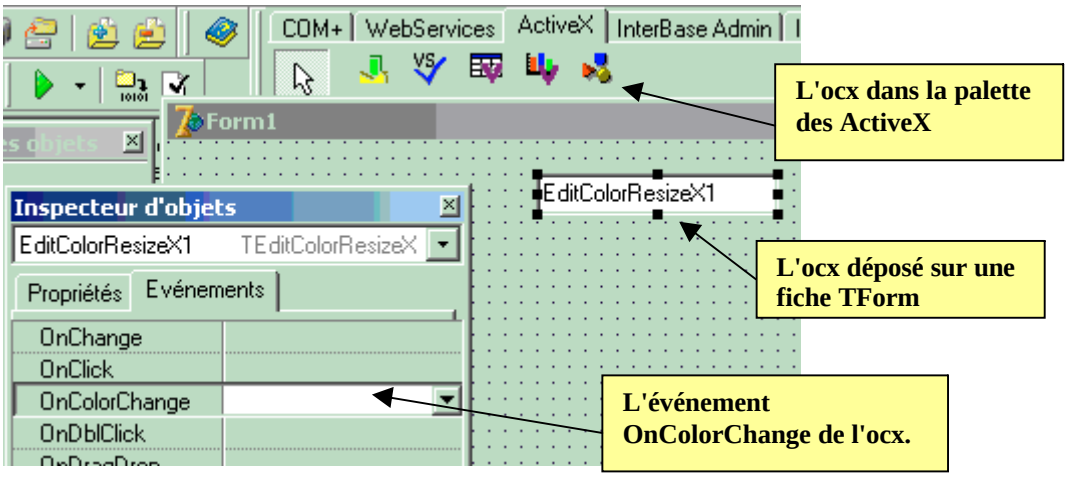
La suite du dialogue avec Delphi est identique à celle que vous avez lorsqu'il s'agit d'installer un composant Delphi classique, le dialogue se termine par l'affichage de la boîte suivante :



A partir de cet instant lorsque vous avez validé l'installation dans la palette vous disposez de la même entité représentée sous deux formats différents :

Un éditeur mono-ligne sensible au changement de couleur et au redimensionnement, vous en avez un version VCL (donc utilisable uniquement avec Delphi) et une autre version ocx utilisable avec tous les environnements de développements qui acceptent les ActiveX.

Enfin ce composant est converti par Net Framework à travers l'outil de conversion :
aximp c:\….\ EditColorResizeXControl1.ocx → en un fichier **AxEditColorResizeXControl1.dll** utilisable par tout langage de l'architecture Net.



4. Déploiement et utilisation Web d'une fiche ActiveX

Delphi propose un expert pour le déploiement web d'une fiche ActiveX :

> L'expert ActiveForm permet de créer de toutes pièces une nouvelle application ActiveX. L'expert configure le projet et ajoute une fiche vide dont on peut commencer la conception en ajoutant des contrôles.

La coclasse TActiveFormX = class(TActiveForm, IActiveFormX) représente une fiche ActiveX, c'est un contrôle ActiveX fondé sur une fiche VCL sur laquelle vous pouvez déposer n'importe lequel des composants de la palette de Delphi d'où un gain de temps de développement.

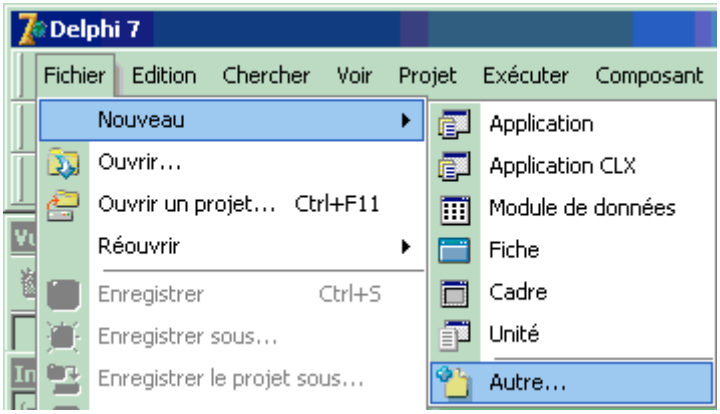
La fiche ActiveX ainsi créée peut être déployée sur le Web : la fiche ActiveForm est ensuite affichable et exécutable à l'intérieur d'une page html dans un navigateur Web (Internet Explorer). Dans le navigateur Web, la fiche se comporte comme une application autonome et peut mettre en œuvre des actions complexes. Les fiches ActiveX sont les "concurrents" (mono-plateforme) des applets Java (multi-plateforme).

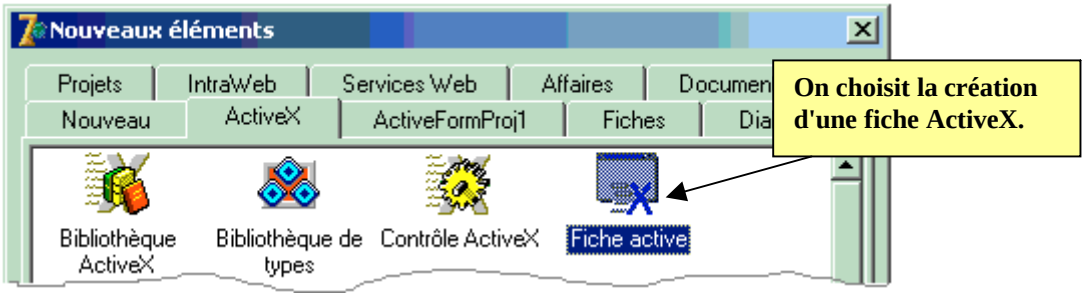
Utiliser l'expert ActiveForm pour créer une Fiche ActiveX basée sur une fiche VCL

La démarche est sensiblement la même que pour créer un contrôle ActiveX, nous allons procéder à une création d'une telle fiche pas à pas sur un exemple.

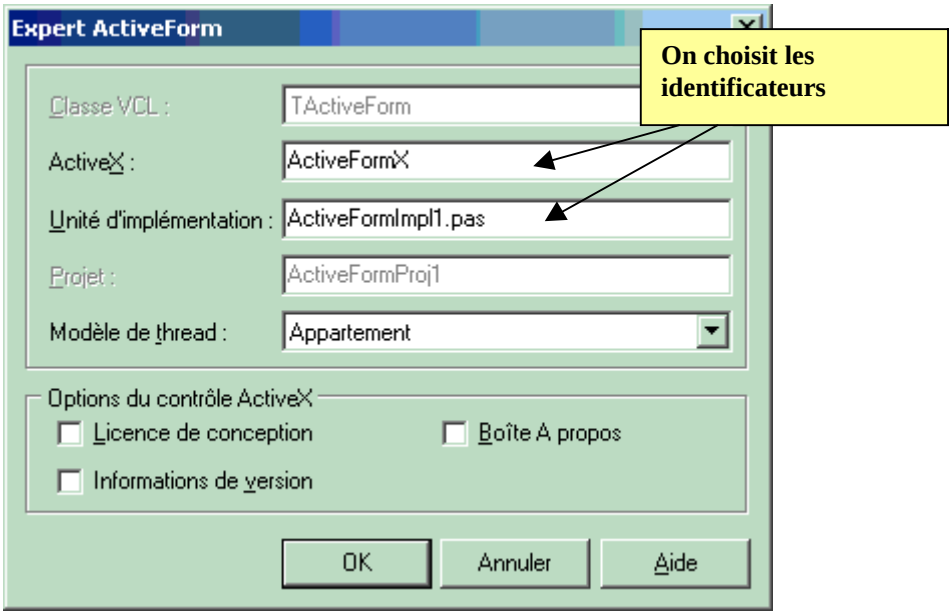
Construisons pas à pas , en 4 étapes, une fiche ActiveX

1°) Construction de la fiche de base

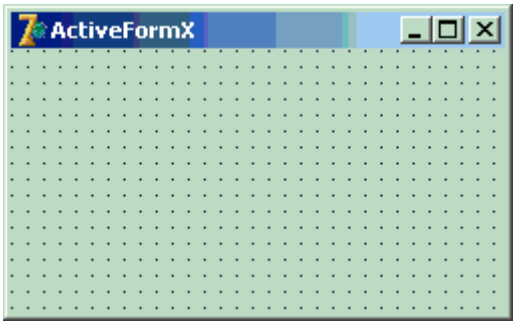
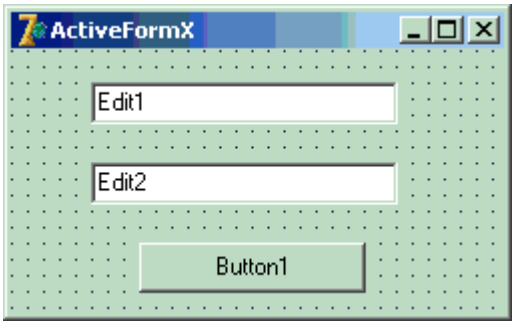




Delphi envoie un formulaire de saisie d'information sur le paramétrage de création du contrôle ActiveX qui hérite obligatoirement de TActiveForm, nous lui indiquons éventuellement le nom de la classe et celui de la unit (nous laissons le modèle de thread par défaut) :



A ce stade, l'expert génère comme au paragraphe précédent, 3 fichiers intermédiaires (de 21 +349 +334 = 704 lignes) et propose une fiche visuelle vierge sur laquelle on peut insérer des contrôles et travailler avec la fiche comme on le ferait pour toute application Delphi.

Fiche ActiveX vierge	Nous déposons deux TEdit et unTButton
	

type
TActiveFormX = **class** (TActiveForm, IActiveFormX)
private
FEvents: IActiveFormXEvents;
procedure ActivateEvent(Sender: TObject);
procedure ClickEvent(Sender: TObject);
procedure CreateEvent(Sender: TObject);
……
	type
	TActiveFormX = **class** (TActiveForm, IActiveFormX)
	Button1: TButton;
	Edit1: TEdit;
	Edit2: TEdit;
	Lignes engendrées
	private
	FEvents: IActiveFormXEvents;
	procedure ActivateEvent(Sender: TObject);
	procedure ClickEvent(Sender: TObject);
	procedure CreateEvent(Sender: TObject); ……

Nous programmons la recopie du texte de l'Edit2 dans l'Edit1 lorsque l'on clique sur le Button1, comme pour une application, Delphi nous propose le squelette du gestionnaire d'événement OnClick du Button1 que nous remplissons avec notre ligne de code de copie :

```
procedure TActiveFormX.Button1Click(Sender: TObject);
begin
   Edit1.Text:=Edit2.Text
end;
```

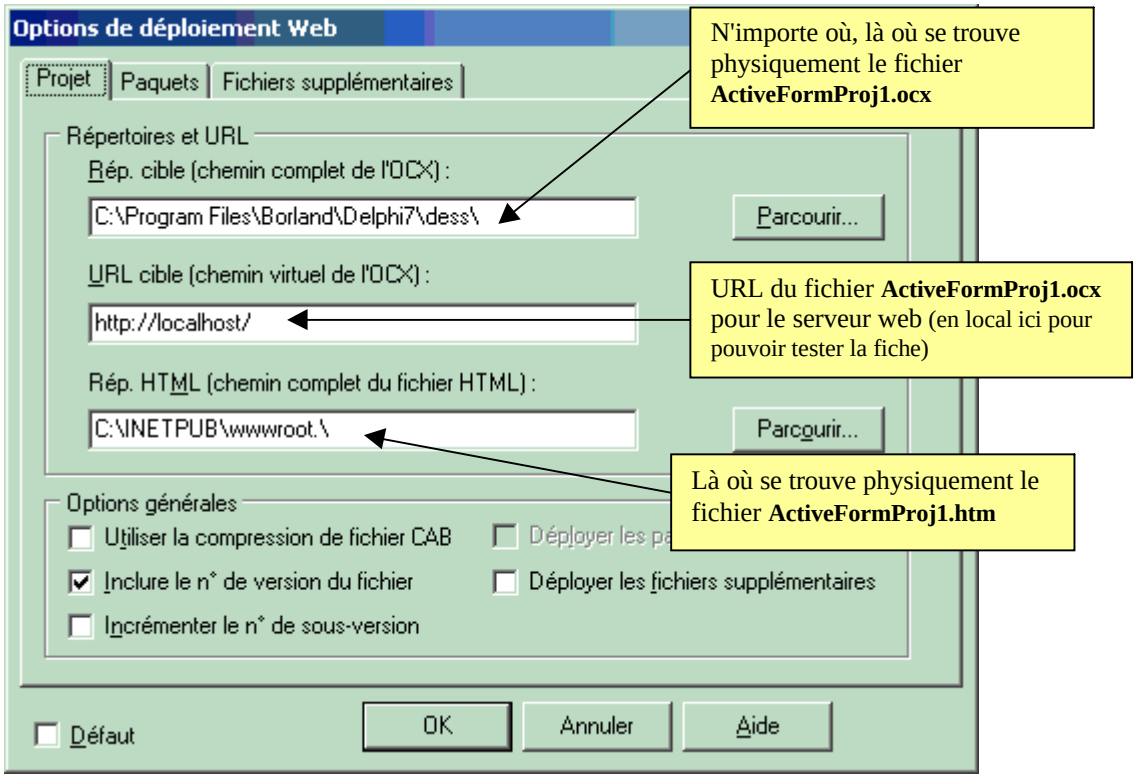
Nous supposons nous arrêter à ce stade et passer à la suite de la création de l'ActiveX.

2°) Recensement de la fiche ActiveX ainsi construite (ActiveFormProj1.ocx)

3°) Déploiement sur le web

Avant de pouvoir utiliser dans un client Web, comme Internet Explorer, la fiche ActiveFormProj1.ocx ainsi créée, il faut la déployer sur le serveur. A chaque fois que la fiche ActiveX est modifiée, elle doit être recensée de nouveau et re-déployée afin que les modifications soient effectives.

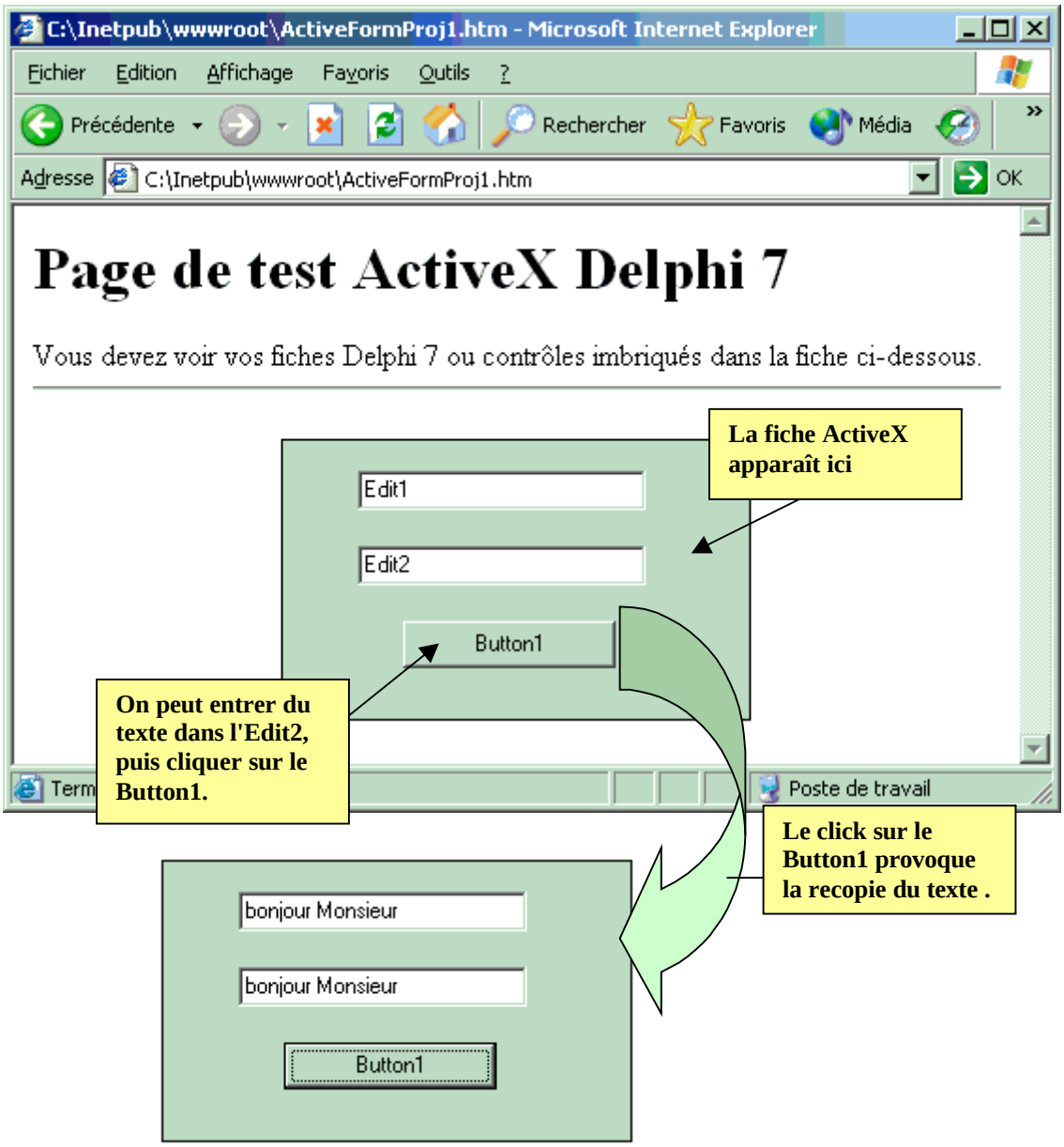
Pour déployer un contrôle ActiveX, il est nécessaire de disposer d'un serveur Web Web (IIS pour notre exemple). Par contre, il n'est pas obligatoire d'enregistrer le contrôle dans la palette de composant puisque, en général, le composant web créé est dédié à une application web particulière.



Delphi engendre un fichier **ActiveFormProj1.htm** permettant de tester la fiche ActiveX :

```
<HTML>
<H1> Page de test ActiveX Delphi 7 </H1><p>
Vous devez voir vos fiches Delphi 7 ou contrôles imbriqués dans la fiche ci-dessous.
<HR><center><P>
<OBJECT
        classid="clsid:F9A089C7-B331-404E-B21F-4541846FA390"
        codebase="http://localhost/ActiveFormProj1.ocx#version=1,0,0,0"
        width=350
        height=250
        align=center
        hspace=0
        vspace=0 >
</OBJECT>
</HTML>
```

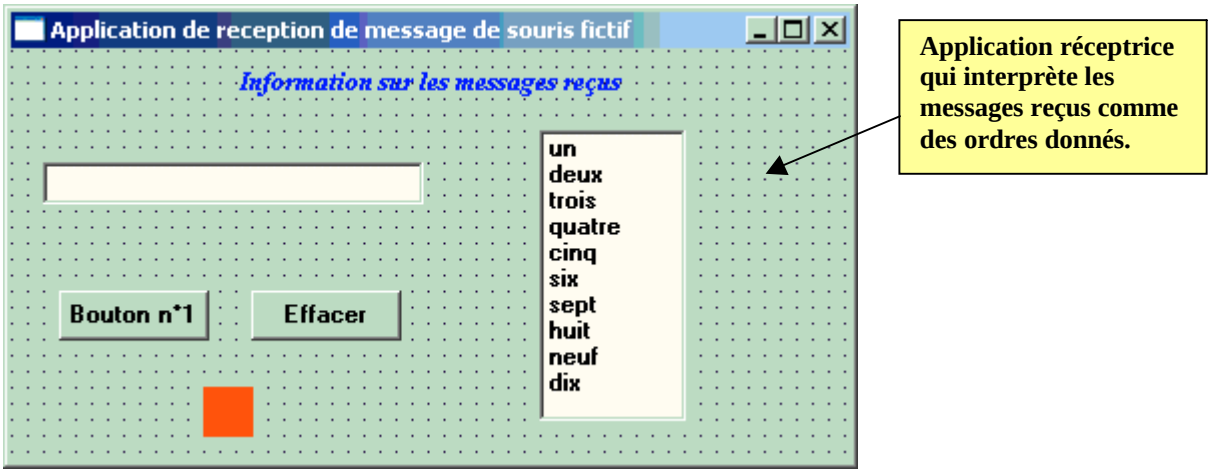
Si vous avez installé un serveur web sur votre machine IIS par exemple alors l'exécution du fichier **ActiveFormProj1.htm** dans le navigateur IE donne la page html de test suivante :



Quand cette page HTML est visualisée dans le navigateur Web, la fiche est affichée et exécutée comme application incorporée dans le navigateur. C'est-à-dire qu'elle s'exécute dans le même processus que le navigateur.

Exercices chapitre 8

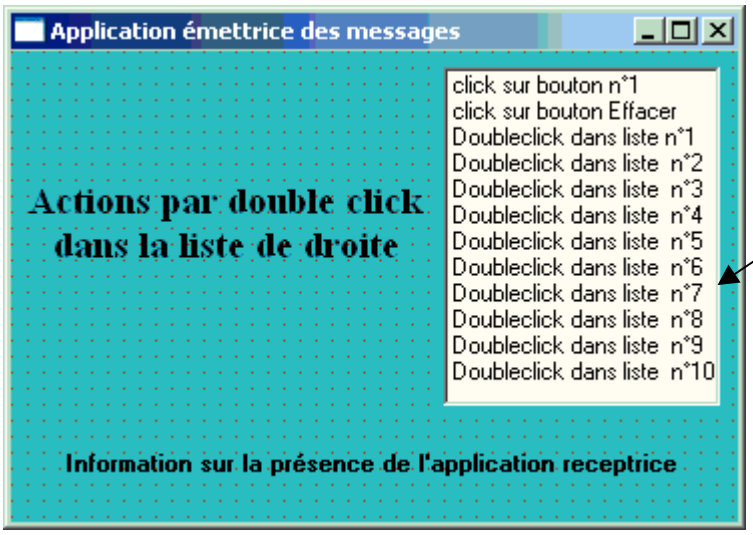
Ex-1 : Exercice entièrement traité sur la prise de contrôle d'une application par une autre à travers des messages systèmes. Le programme comporte une application de reception de messages et une application émettant des messages vers l' application de reception. L'application émettrice envoie des messages qui sont interprétés par la réceptrice comme des manipulations de la souris locale (déplacement, click, double click).




```
unit Unit1;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

 type
  TFAppliRecept = class(TForm)
  Button1: TButton;
  ListBox1: TListBox;
  Edit1: TEdit;
  Button2: TButton;
  LabelInfo: TLabel;
  LabelVisu: TLabel;
  procedure FormCreate(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  procedure ListBox1DblClick(Sender: TObject);
  procedure ButtonEffacerClick(Sender: TObject);
  procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
   Y: Integer);
 private
 { Déclarations privées }
  procedure MessageUser(var mess:TMsg; var hand:boolean);
 public
 { Déclarations publiques }
end;

 var
  FAppliRecept: TFAppliRecept;

implementation
```

```
{$R *.dfm}

Const
 WM_ClickBouton1 = WM_User; // message envoyé = WM_USER (<=>interprété click bouton n°1)
 WM_ClickEffacer = WM_User+1; // message envoyé = WM_USER+ 1(<=>interprété click bouton effacer)
 wm_dblclickliste = wm_user+2;
 {message envoyé = wm_user+ 2 (<=>interprété double click dans la liste)
                   wparam contient le rang de l'élément sélectionné dans la liste
 }
 wm_mousesurfiche = wm_user+3;
 {message envoyé = wm_user+ 3 (<=>interprété déplacement souris sur la fiche)
                   WParam contiendra la coordonnée X de la souris
                   lparam contiendra la coordonnée y de la souris
 }


 {--- le gestionnaire de traitement du OMessage de l'application ---}

 procedure TFAppliRecept.MessageUser(var mess:TMsg;var hand:boolean);
 { traite le message wm_User intercepté comme un ordre donné à la fenêtre.
   il est traité à partir du niveau onmessage de l'application
 }
 begin
  if mess.message=WM_ClickBouton1 then // message envoyé = WM_USER ici
  begin
   LabelInfo.Caption:='WM_ClickBouton1 : simulant un click sur bouton n°1';
   Button1.Click;
  end
  else
   if mess.message=wm_clickeffacer then  // message envoyé = wm_user+1 ici
   begin
    LabelInfo.Caption:='WM_ClickEffacer : simulant un click sur bouton Effacer';
    Button2.Click;
   end
   else
    if mess.message=wm_dblclickliste then
    begin
     LabelInfo.Caption:='WM_DblClickListe : simulant un double click sur la liste';
     ListBox1.ItemIndex:=mess.wParam; //le rang de l'élément sélectionné
     ListBox1DblClick(ListBox1)
    end
    else
     if mess.message=wm_mousesurfiche then
     begin
      LabelInfo.Caption:='WM_MouseSurFiche : simulant la position de la souris sur la fiche';
      FormMouseMove(self, [ ], mess.wParam, mess.lParam)
     end;
  inherited; //laisse delphi s'occupper des autres messages
 end;

 {------------------ LES GESTIONNAIRES D'EVENMENTS --------------------}

 procedure TFAppliRecept.FormCreate(Sender: TObject);
 //connexion du gestionnaire de OnMessage de l'application
 begin
  Application.OnMessage:=MessageUser;
 end;
```

```
{---- Les gestionnaires d'évènements qui seront appelés par l'émétteur
et qui fonctionnent déjà en local lorsque l'appli est activée ----
}

procedure TFAppliRecept.Button1Click(Sender: TObject);
//gestion du click du button1
begin
 Edit1.text:='Le bouton n° 1 vient d"être clické';
end;

procedure TFAppliRecept.ButtonEffacerClick(Sender: TObject);
//gestion du click du button2
begin
 Edit1.Clear
end;

procedure TFAppliRecept.ListBox1DblClick(Sender: TObject);
//gestion du double click du ListBox1
begin
 with Sender as TListBox do
  Edit1.text:='choix dans la liste : '+ListBox1.Items[itemindex];
end;

procedure TFAppliRecept.FormMouseMove(Sender: TObject; Shift: TShiftState;
 X, Y: Integer);
 //gestion du mousemove sur la fiche
 begin
  Edit1.text:='souris/ x = '+inttostr(x)+' y = '+inttostr(y);
  LabelVisu.Top:=Y;
  LabelVisu.Left:=X;
 end;

end.
```



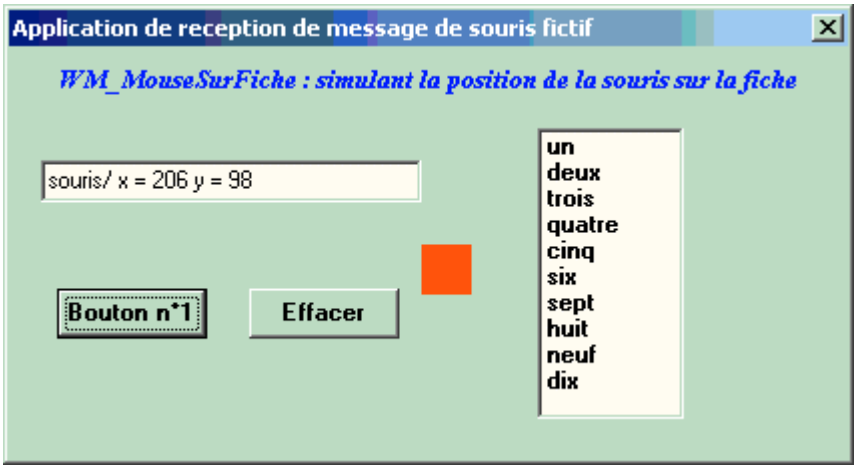
Application émettrice qui envoie les messages comme des ordres donnés à la souris de l'application réceptrice.

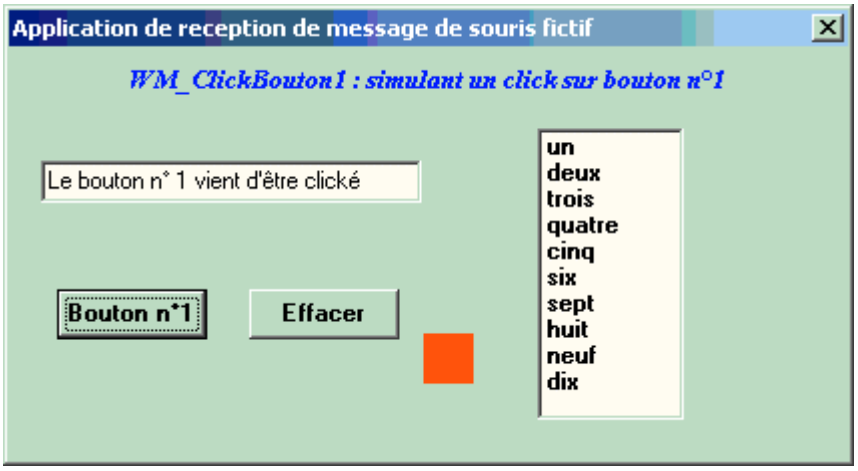
```
unit Unit1;
{    cette application contrôle l'action de la souris dans une autre
application au niveau du click sur un bouton, double clik et mouse move.
```

```
le nom de la classe TForm de la fenêtre de l'appli réceptrice est TFAppliRecept.
On construit 4 messages utilisateurs :
WM_ClickBouton1 = WM_User;
WM_ClickEffacer = WM_User+1;
WM_DblClickListe = WM_User+2;
WM_MouseSurFiche = WM_User+3;

et PostMessage est chargée de les expédier à  TFAppliRecept (avec des paramètres éventuels)

}
interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

 type
  TForm1 = class(TForm)
  LabelInfoMess: TLabel;
  ListBoxMess: TListBox;
  Label1: TLabel;
  procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
   Y: Integer);
  procedure ListBoxMessDblClick(Sender: TObject);
 private
 { Déclarations privées }
 public
 { Déclarations publiques }
end;

 var
  Form1: TForm1;

implementation

 {$R *.dfm}

 procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
 var
  Wnd:HWnd;
  begin
   Wnd:=FindWindow('TFAppliRecept',nil);// recherche de la fenêtre
   if Wnd<>0 then// si instance de la fenêtre  trouvée
   begin
    LabelInfoMess.Caption:='Fenêtre TFAppliRecept trouvée';
    PostMessage(Wnd,WM_USER+3,X,Y);// WM_MouseSurFiche
    {les paramètres X et Y donnent les coord. de la sourisà utiliser
    dans la fenêtre réceptrice
    }
   end
  end;


 procedure TForm1.ListBoxMessDblClick(Sender: TObject);
 var
  Wnd:HWnd;
  begin
   Wnd:=FindWindow('TFAppliRecept',nil);// recherche de la fenêtre
   if Wnd<>0 then// si instance de la fenêtre  trouvée
```

```
 begin
  LabelInfoMess.Caption:='Fenêtre TFAppliRecept trouvée';
  case
   listboxmess.itemindex of
   0:PostMessage(Wnd,WM_USER,0,0); //WM_ClickBouton1
   1:PostMessage(Wnd,WM_USER+1,0,0);// WM_ClickEffacer
   2..11:PostMessage(Wnd,WM_USER+2,ListBoxMess.itemindex-2,0);// WM_DblClickListe
   {la valeur du paramètre ListBoxMess.itemindex-2 indique le rang de
       l'élément à clicker dans la liste de la fenêtre de réception
   }
  end;
 end
end;

end.
```

Exemples

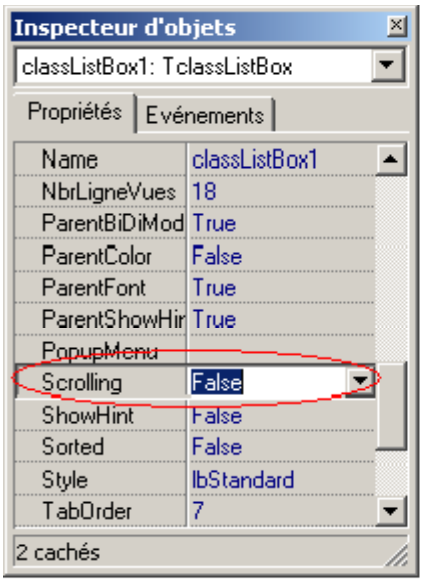


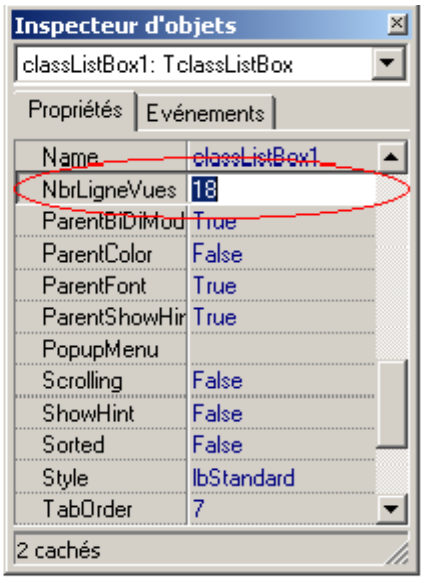


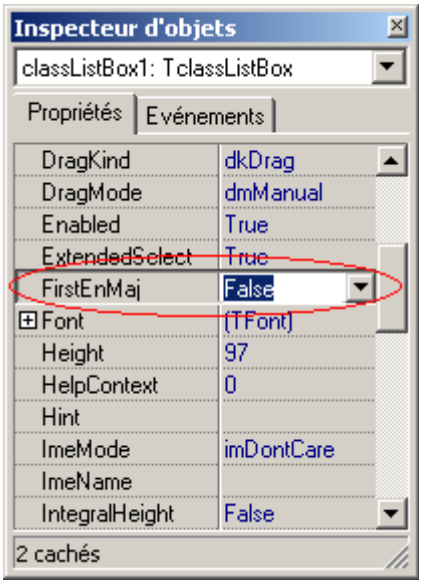
etc…

Ex-2 : Complétez la Unit suivante de telle manière que la classe TclassListBox représente un composant héritant des TListBox avec les améliorations suivantes :

Trois nouvelles propriétés :
Scrolling:boolean *{permettant le défilement automatique du texte}*
NbrLigneVues:integer *{permettant de fixer le nombres de lignes de texte affichées}*
FirstEnMaj:boolean *{permettant d'autoriser la mise en majuscule du premier caractère de tous les lignes de la liste}*

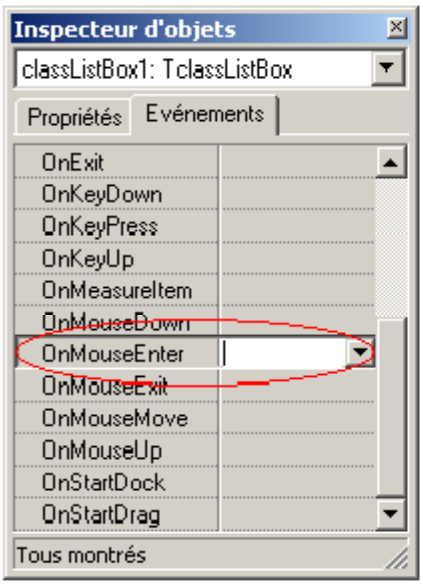


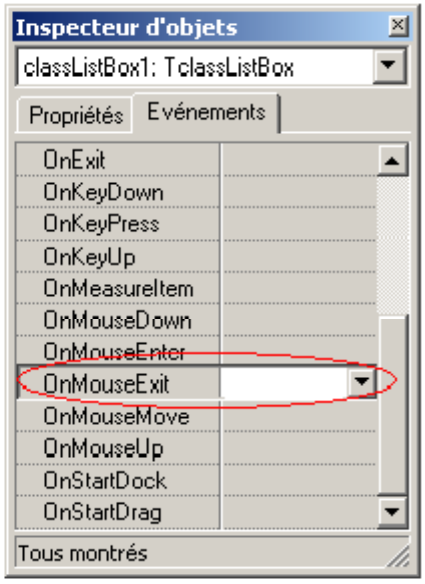




Deux nouveaux événements :

OnMouseEnter: *{interceptant l'entrée de la souris dans le composant}*
OnMouseExit: *{interceptant la sortie de la souris hors du composant}*





Code fourni : Squelette de la Unit à compléter

```
unit UclassListBox;

interface
uses stdctrls,Messages, Controls,Windows,Classes,Sysutils,extctrls;

type

 TclassListBox=class(TListBox)
   private
    FOnMouseEnter: TNotifyEvent;
    FOnMouseExit: TNotifyEvent;
    FFirstEnMaj:boolean;// champ d'autorisation de mise en majuscule du 1er caractère
    FNbrLigneVues:integer;// champ indiquant le nombre de lignes affichables dans la liste
    FScrolling:boolean; // champ d'autorisation de défilement automatique
```

```
    Tempo:TTimer;// Timer de défilement ligne à ligne de la liste
    procedure SetMaj(x:boolean);
    procedure CMMOUSEENTER(var mess:TMessage);message CM_MOUSEENTER; // VCL borland
    procedure CMMOUSELEAVE(var mess:TMessage);message CM_MOUSELEAVE ; // VCL borland
    procedure LBADDSTRING(var mess:TMessage);message  LB_ADDSTRING; //system
    procedure WMSIZE(var mess:TMessage);message WM_SIZE; //system
    procedure setScrolling(x:boolean);
    procedure setNbrLigneVues(x:integer);
    procedure OnTempo(Sender:TObject);
   public
    constructor Create(AOwner: TComponent); override;
   published
    property NbrLigneVues:integer read FNbrLigneVues write setNbrLigneVues;
    property Scrolling:boolean read FScrolling write setScrolling;
    property FirstEnMaj:boolean read FFirstEnMaj write SetMaj;
    property OnMouseEnter: TNotifyEvent read FOnMouseEnter write FOnMouseEnter;
    property OnMouseExit: TNotifyEvent read FOnMouseExit write FOnMouseExit;

 end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents( 'Exemples', [TclassListBox]);
end;

{ TclassListBox }
constructor TclassListBox.Create(AOwner: TComponent);
//le constructeur du composant
begin

end;


procedure TclassListBox.WMSIZE(var mess: TMessage);
{lors du changement de taille du composant :
   - le nombre de lignes vues change,
   - la première ligne du texte doit être affichée au début
   - le défilement est éventuellement arrêté selon le nombre de lignes affichées
}
begin

end;

{----------- entrée et sortie de la souris -----------------}
procedure TclassListBox.CMMOUSEENTER(var mess: TMessage);
//lance le gestionnaire d'événement OnmouseEnter
begin

end;

procedure TclassListBox.CMMOUSELEAVE(var mess: TMessage);
//lance le gestionnaire d'événement OnmouseExit
begin

end;
```

```
{----------- le premier caractère d'une ligne  -----------------}
procedure TclassListBox.LBADDSTRING(var mess: TMessage);
{lorsqu'une nouvelle ligne est insérée dans la liste, le premier caractère est mis en majuscule
 (si le composant est autorisé à mettre en majuscule).}
begin

end;

procedure TclassListBox.SetMaj(x: boolean);
{Met en majuscule tous les premiers caractères de chaque ligne si x est true,  sinon met tous
ces premiers caractères en minuscule et positionne FirstEnMaj:boolean  à la valeur adéquate.}
begin

end;

{-----------le défilement automatique -----------------}
procedure TclassListBox.OnTempo(Sender: TObject);
{le défilement ligne à ligne de toute la liste jusqu'à la fin  en boucle si le nombre ligne de la liste est plus grand
que le  nombre de ligne affichables (NbrLigneVues:integer donne ce nombre)
et si le défilementest autorisé (Scrolling:boolean donne cette autorisation)
}
begin

end;

procedure TclassListBox.setScrolling(x: boolean);
{positionne le défilement ligne à ligne de toute la liste jusqu'à la fin
 en boucle si le nombre ligne de la liste est plus grand que le
 nombre de ligne affichables (NbrLigneVues:integer donne ce nombre)
}
begin

end;

procedure TclassListBox.setNbrLigneVues(x: integer);
{positionne le nombre de ligne affichables (NbrLigneVues:integer)
 recalcule et modifie la hauteur du composant en fonction du nombre de lignes
 que l'on veut afficher.
}
begin

end;

end.
```

Messages à utiliser (conseils) :

{

CM_MOUSELEAVE , CM_MOUSEENTER : pour la souris
LB_ADDSTRING , LB_GETTOPINDEX , LB_SETTOPINDEX : pour les ListBox
WM_SIZE : pour le redimensionnement du composant
WM_VSCROLL : pour la barre de défilement vertical

}

2°) Fournissez le composant prêt à installer .

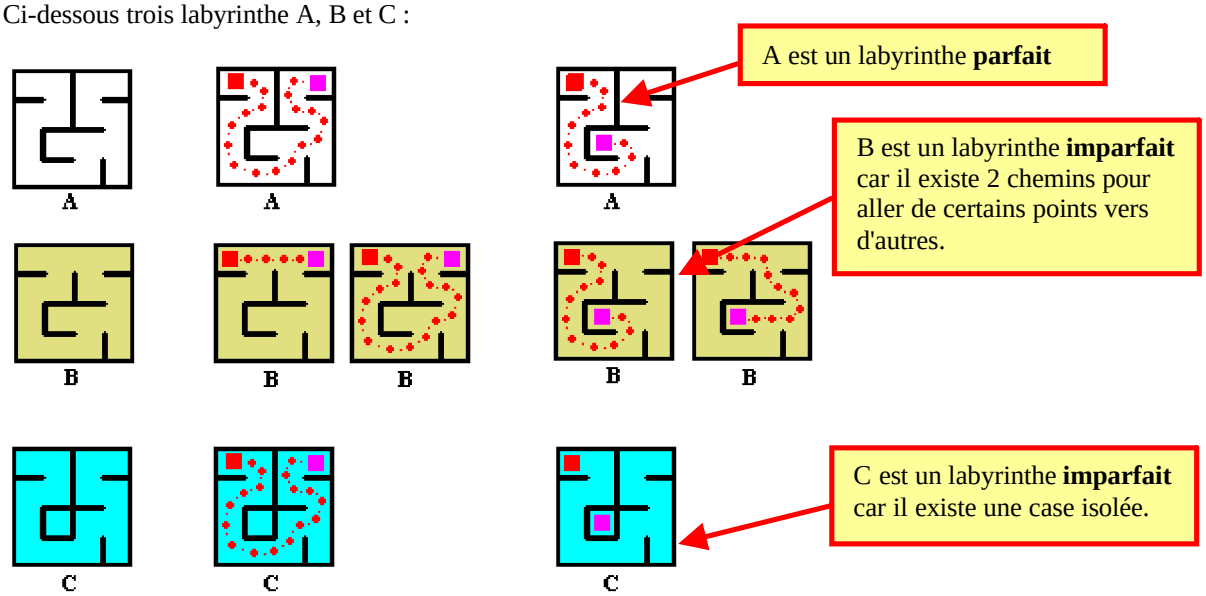
Ecrivez un programme Delphi de test du composant.

Ex-3 : non déterminisme et retour arrière algorithme d'exploration d'un labyrinthe

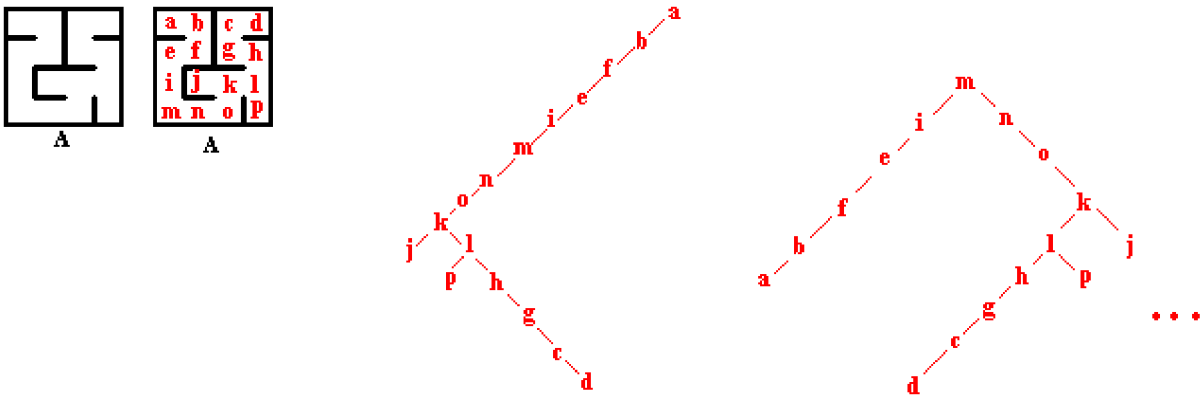
Un labyrinthe est représenté par un tableau de cases, c'est un labyrinthe **parfait** si :

> Deux cases quelconques sont joignables
> par un chemin unique.il n'existe pas de case isolée.

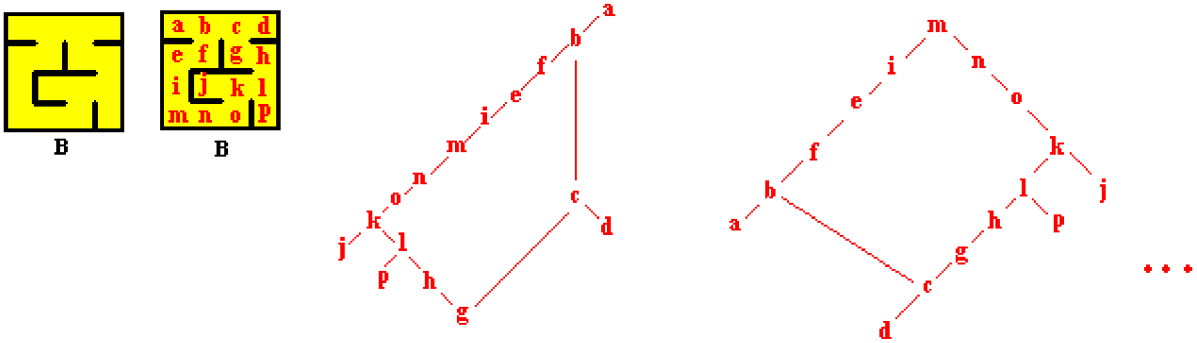
Ci-dessous trois labyrinthe A, B et C :



Les parcours dans un labyrinthe **parfait** peuvent être représentés par des arbres dont chaque noeud a au plus 3 fils :



Dans un labyrinthe **imparfait** les parcours sont représentés par des graphes non arborescents :



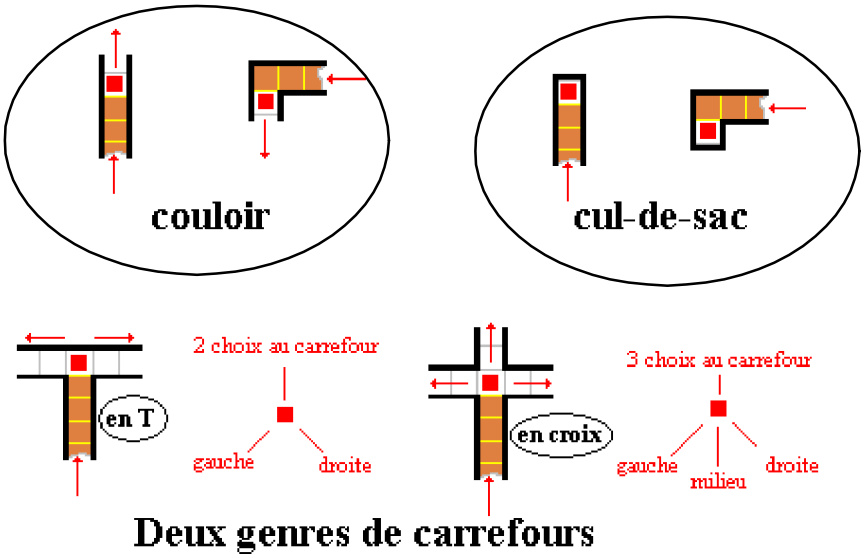
imparfait

Objectif :

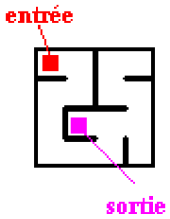
Nous allons faire explorer un labyrinthe semi-parfait (il peut y avoir plusieurs chemins entre deux cases, mais il n'y a aucune case isolée) à un **pélerin,** il aura pour mission de trouver une sortie s'il y en a une, ou bien de nous assurer qu'après exploration complète de ce labyrinthe, il ne comporte pas de sortie.

Pour notre pélerin un labyrinthe à explorer est caractérisé par une **entrée,** une succession de **couloirs**, de **cul-de-sacs** et de **carrefours** (et éventuellement une sortie).

Ci-dessous les configurations topographiques associées :



Dans un labyrinthe parfait, nous sommes assurés que s'il existe une sortie, il existe alors un chemin unique allant de la case marquée **entrée** vers la case marquée **sortie**.



Le travail de notre pélerin va être de chercher ce chemin s'il existe, par exploration exhaustive.

On apprend son "métier" au pélerin

Nous ne voulons pas que notre pélerin se perde dans le labyrinthe en explorant plusieurs fois le même chemin aussi va-t-on lui enseigner l'**art étrange** du parcours en profondeur avec retour arrière (back-tracking disent les anglophones) sans repasser sur un chemin déjà exploré. Notre centre de formation de pélerin explorateur de labyrinthe ne reculant devant aucune dépense, munit notre voyageur d'une lampe torche à durée de vie importante, de deux sacs chacun remplis de cailloux, l'un de cailloux roses l'autre de cailloux gris.

L'instructeur lui apprend qu'il va suivre des couloirs dans le labyrinthe, qu'il va changer de direction à des carrefours pour emprunter un nouveau couloir, qu'il va aussi tomber sur des culs-de-sacs. On lui apprend qu'il va devoir changer son comportement dès qu'il rencontre un cul-de-sac.

Tantqu'il chemine dans un couloir on lui conseille de déposer derrière lui un caillou rose, s'il arrive à un carrefour il dépose toujours un caillou rose et emprunte un des couloirs disponibles.

Dès qu'il rencontre un cul-de-sac, il doit évidemment revenir sur ses pas en ramassant les cailloux roses qu'il

a semés, lorqu'il arrive en revenant sur ses pas, à un carrefour, il dépose derrière lui un caillou gris à l'entrée du couloir qu'il est en train de quitter. Il doit aller au milieu du carrefour (qui contient un caillou rose) et il examine le seuils des différents couloirs qui débouchent sur ce carrefour :

- Un seul couloir a un caillou rose sur son seuil, les autres couloirs qu'il a déjà explorés ont un caillou gris sur leur seuil, ceux qu'il n'a pas encore explorés n'ont aucun caillou sur leur seuil.
- Dans l'éventualité où tous les seuils ont un caillou, il doit impérativement ramasser le caillou rose du carrefour et emprunter l'unique couloir dont le seuil est marqué par un caillou rose, tout en ramassant les cailloux rose semés dans ce couloir (il vient en fait d'explorer tous les chemins qui partaient de ce carrefour)

L'examen des seuils doit se faire par observation des cases (des entrées de couloir), le pélerin doit connaître le nombre de cases voisines de celle sur laquelle il se trouve, il en déduira le type de situation dans laquelle il est :

Résultat de l'examen des cases voisines	Situation dans laquelle se trouve le pélerin
1 case voisine	Le pélerin est actuellement dans un couloir.
aucune case voisine	Le pélerin est actuellement dans un cul-de-sac.
2 cases voisines	Le pélerin est actuellement sur un carrefour à 2 choix.
3 cases voisines	Le pélerin est actuellement sur un carrefour à 3 choix.

Bien entendu on a indiqué au pélerin que dès qu'il trouvait une sortie, il devait l'emprunter et son voyage se terminait là.

Algorithme de parcours en profondeur

Nous proposons une version itérative et une version récursive du parcours dans le labyrinthe.

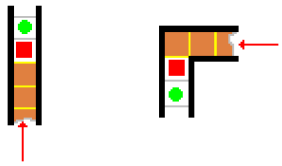
Version itérative

Algorithme Labyrinthe
*{ Un algorithme de déplacement dans un labyrinthe parfait. Version **ITERATIVE** }*
global
le labyrinthe
sens € Booleen *// indique le sens actuel de parcours*
SortieFound € Booleen *// une sortie a été trouvée*
utilise
DeplacerAller *// déplacement d'une case dans le sens exploration*
DeplacerEnRetour *// déplacement d'une case en revenant sur ses pas*

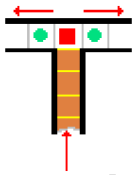
debut
tantque non SortieFound **faire**
si sens = aller **alors**
DeplacerAller
sinon
DeplacerEnRetour
fsi
ftant
fin *// Labyrinthe*
Algorithme DeplacerAller
{ Cheminement en mode "aller" à travers des couloirs et des carrefours : Actions lorsque le pélerin est sur une case du labyrinthe.
}
global
le labyrinthe
sens € Booleen *// indique le sens actuel de parcours*
SortieFound € Booleen *// une sortie a été trouvée*

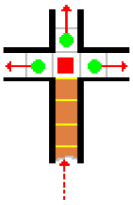
debut
si la case est une **sortie alors**
SortieFound <-- **vrai ;**
arrêt de l'exploration
sinon
Examen des cases voisines **;**
si aucune case voisine **alors** *//cul-de-sac*

aucune case voisine
{ il change de sens et se prépare è revenir sur ses pas }
sens <-- retour **;**
le pélerin ramasse le caillou rose **;**
sinon
si une seule case est voisine **alors** *//couloir*



1 case voisine
le pélerin se déplace **vers** cette case **;**
si la case actuelle n'est pas l' **Entrée alors**
il laisse un caillou rose sur la case actuelle **;**
fsi
sinon *//carrefour (2 ou 3 voisins)*





2 cases voisines • , **3 cases voisines** •

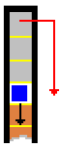
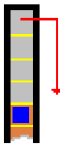
le pélerin choisit aléatoirement une case libre **;**
le pélerin se déplace **vers** cette case **;**
si la case actuelle n'est pas l' **Entrée alors**
il laisse un caillou rose sur la case actuelle **;**
fsi
fsi
fsi
fin *// DeplacerAller*

Algorithme DeplacerRetour

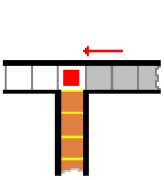
{ Cheminement en mode "retour" à travers des couloirs et des carrefours : Actions lorsque le pélerin est sur une case du labyrinthe. }

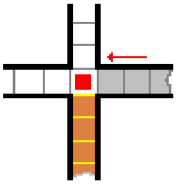
global
le labyrinthe
sens € Booleen *// indique le sens actuel de parcours*

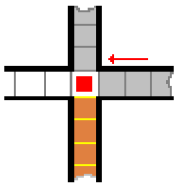
debut
si le pélerin est dans un couloir **alors**

il avance sur la case libre suivante **;**	
il dépose un caillou gris sur la case qu'il vient de quitter **;**	

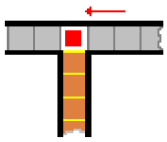
sinon *// la case est un carrefour*
évaluation du nombre de case voisines libres **;**
si au moins une case est libre **alors**

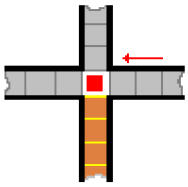






le pélerin avance sur cette case libre
sinon *// toutes les cases voisines ont été explorées*
le pélerin retourne en empruntant la case qui contient
le caillou rose *// l'entrée du couloir de retour*





fsi
fsi

```
fin  // DeplacerRetour
```

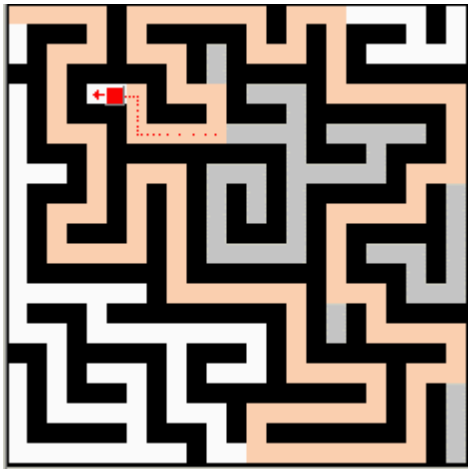
Version récursive

Nous donnons une version récursive qui reprend exactement la stratégie précédente, en particulier les deux algorithmes DeplacerAller et DeplacerEnRetour, en changeant l'itération **tantque** en récursivité (comparez les deux solutions) :

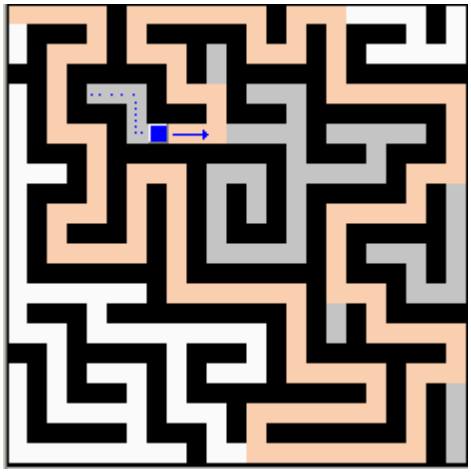
```
Algorithme  Labyrinthe
{ Un algorithme de déplacement dans un labyrinthe parfait. Version RECURSIVE }
global
  le labyrinthe
  SortieFound  € Booleen  // une sortie a été trouvée
  debut
    si  FSortieFound = faux  alors
      seDeplacer
    sinon
      ecrire('La sortie a déjà été trouvée !')
    fsi
 fin  //   Labyrinthe

Algorithme  SeDeplacer
{ le bloc  RECURSIF }
global
  le labyrinthe
  SortieFound  € Booleen  // une sortie a été trouvée
utilise
  DeplacerAller  // déplacement d'une case dans le sens exploration
  DeplacerEnRetour  // déplacement d'une case en revenant sur ses pas
 debut
   si  sens = aller  alors
     DeplacerAller
    sinon
     DeplacerEnRetour
    fsi ;
    SeDeplacer
 fin  //   SeDeplacer
```

Implanter en Delphi les deux versions de cet algorithme. Ci-dessous l'exécution d'un programme Delphi gérant d'une manière animée, le parcours d'un pèlerin (un carré) dans un labyrinthe.

Mode aller :

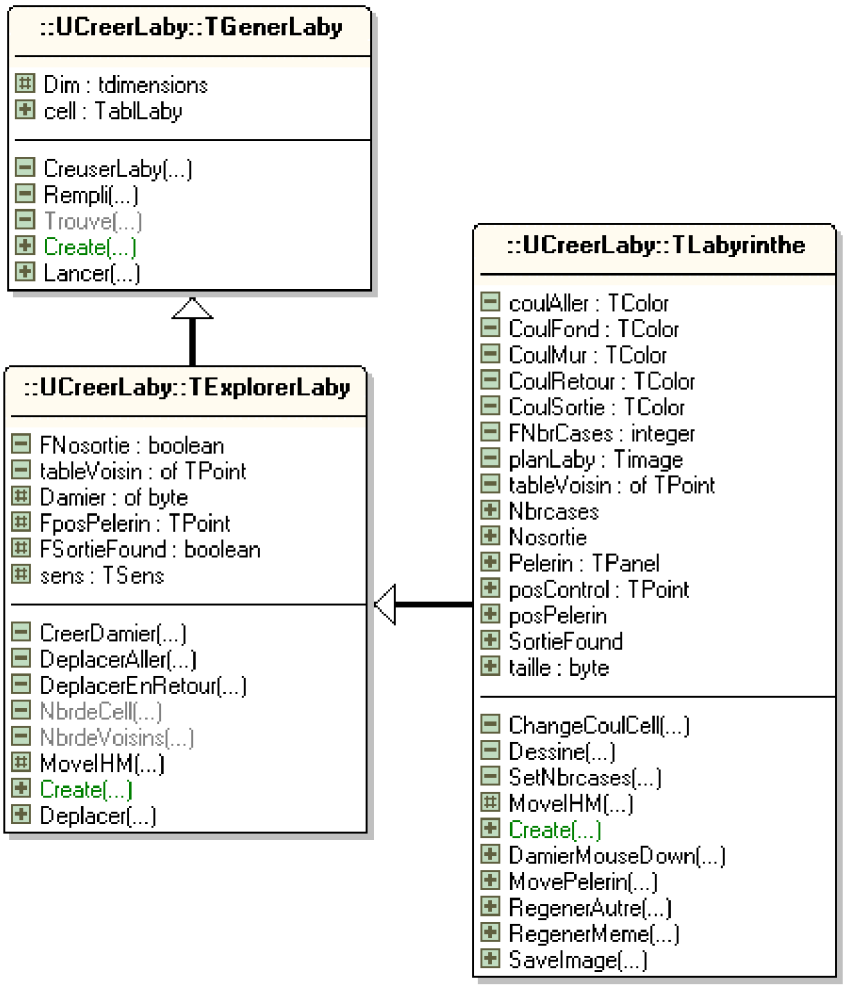


Mode retour :



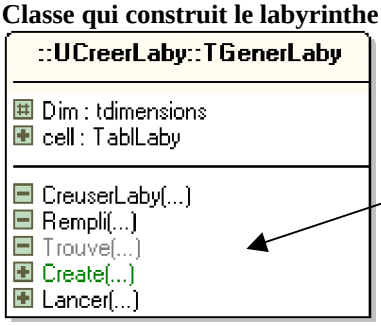
Ex-3 Code solution pratique : Labyrinthe

Une solution d'animation d'un parcours dans un labyrinthe sous forme de 3 classes implémentant un explorateur et un labyrinthe :



L'algorithme de création aléatoire d'un labyrinthe a été trouvé sur Internet (R.Jolivet), plusieurs autres algorithmes sont proposés, le lecteur peut changer d'algorithme pour obtenir des labyrithes de topologies différentes, il lui suffit de remplacer la méthode Lancer de la classe TGenerLaby.

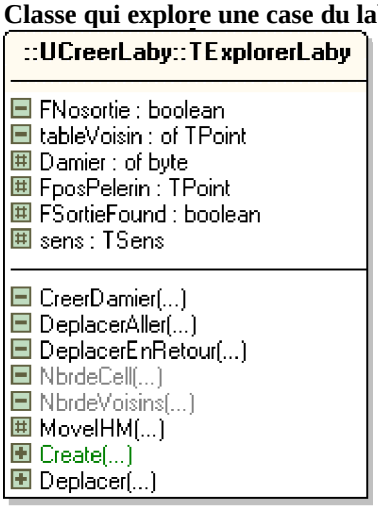
Classe qui construit le labyrinthe



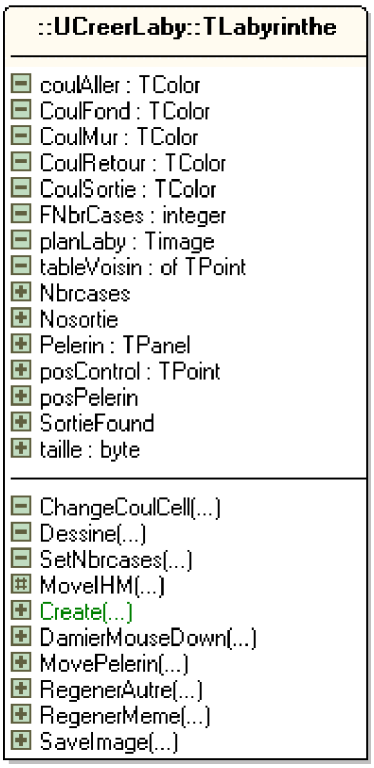
Méthode à changer si vous avez un autre programme de génération de labyrinthes :

```
procedure TGenerLaby.Lancer;
  begin
    Rempli;
    CreuserLaby(0,0,2);
  end;
```

Classe qui explore une case du labyrinthe



Classe dédiée à l'affichage visuel

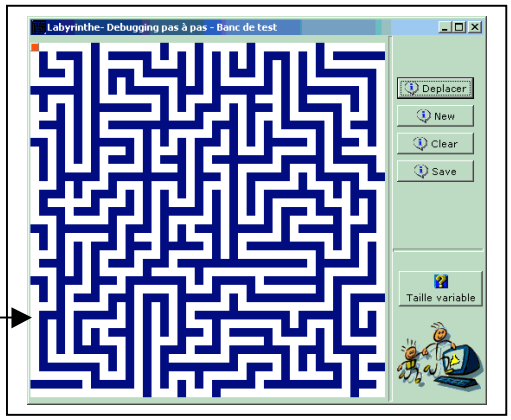


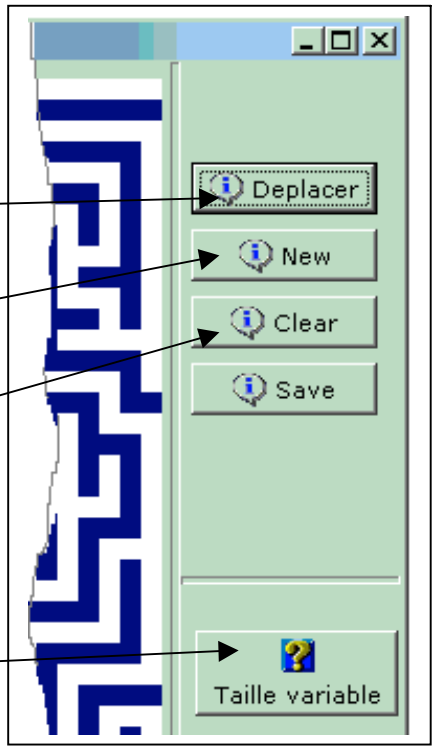
Le programme engendre un labyrinthe aléatoire, puis lance un TTimer qui fait explorer les cases successivement selon l'algorithme de l'énoncé (parcours en profondeur) et affiche visuellement les différents trajets. Le pélerin est un carré de couleur rouge, les cailloux roses sont représentés par une case colorée en rose, les murs sont noirs, les cases libres sont blanches, les cailloux gris sont représentés par une case colorée en gris. Pour bien indiquer visuellement le sens de parcours, le pélerin est de couleur rouge lorsqu'il est en mode aller dans un couloir et en bleu lorsqu'il est en mode retour.

Ci-après une IHM de test du labyrinthe pas à pas

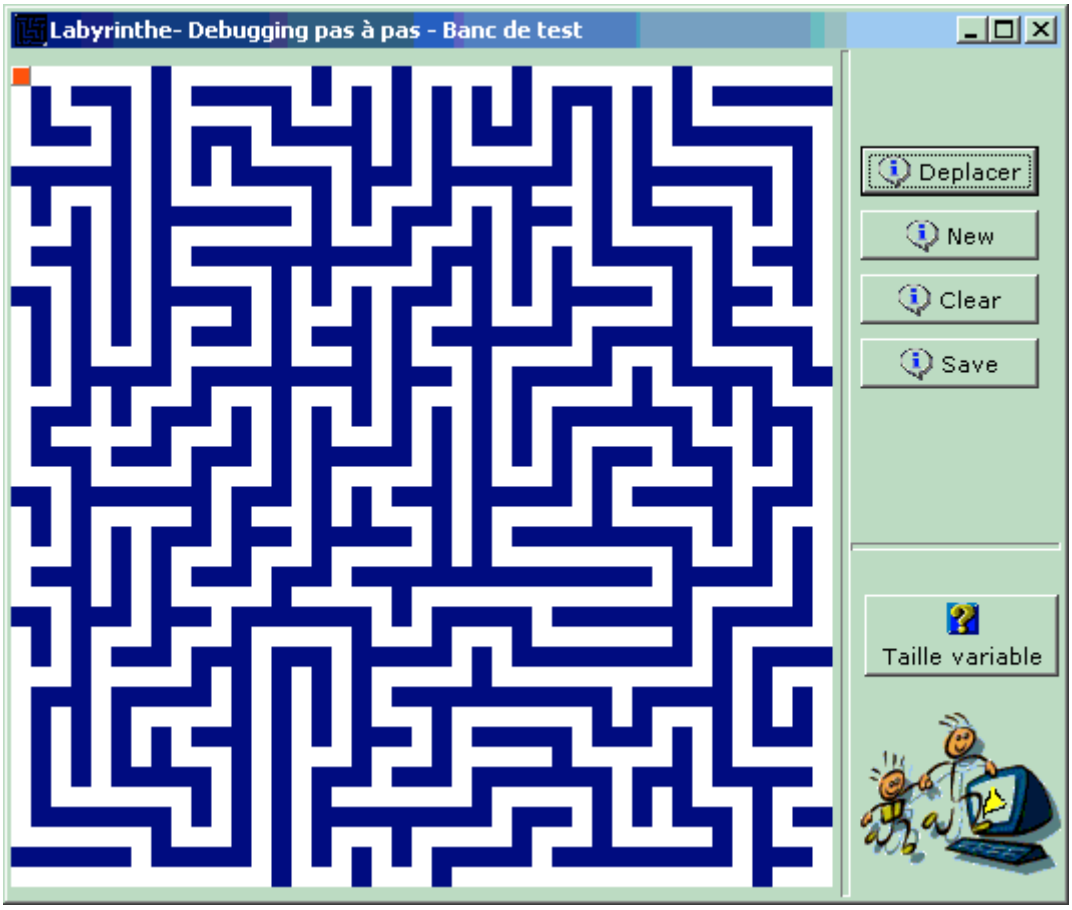
```
unit UFLaby;

 interface

 uses
  UCreerLaby,unit2,
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, ComCtrls, StdCtrls, Buttons, Spin, Menus;

 type
  TForm1 = class(TForm)
  Couleur: TColorDialog;
  SaveDialog1: TSaveDialog;
  Image1: TImage;
  PanelPet: TPanel;
  BitBtnPerso: TBitBtn;
  BitBtnNew: TBitBtn;
  BitBtnClear: TBitBtn;
  BitBtn2: TBitBtn;
  BitBtnsave: TBitBtn;
  Bevel1: TBevel;
```

```
  Bevel2: TBevel;
  Image2: TImage;
  procedure FormCreate(Sender: TObject);
  procedure BitBtnPersoClick(Sender: TObject);
  procedure BitBtnNewClick(Sender: TObject);
  procedure BitBtnClearClick(Sender: TObject);
  procedure BitBtn2Click(Sender: TObject);
  procedure BitBtnsaveClick(Sender: TObject);
 public
 { Déclarations publiques }
 LabyRMD:TLabyrinthe;
end;

 var  Form1: TForm1;

implementation

 {$R *.DFM}

 procedure TForm1.FormCreate(Sender: TObject);
 begin
  LabyRMD:=TLabyrinthe.Create(Image1);
  PanelPet.Width:= LabyRMD.taille;
  PanelPet.Height:= LabyRMD.taille;
  LabyRMD.Pelerin:=PanelPet;
  LabyRMD.posPelerin:=Point(0,0);
  LabyRMD.MovePelerin(Point(0,0));
 end;


 procedure TForm1.BitBtnPersoClick(Sender: TObject);
 begin
  LabyRMD.Deplacer
 end;

 procedure TForm1.BitBtnNewClick(Sender: TObject);
 begin
  LabyRMD.RegenerAutre
 end;

 procedure TForm1.BitBtnClearClick(Sender: TObject);
 begin
  LabyRMD.RegenerMeme
 end;

 procedure TForm1.BitBtn2Click(Sender: TObject);
 begin
  form2.show
 end;

 procedure TForm1.BitBtnsaveClick(Sender: TObject);
 begin
  LabyRMD.SaveImage;
 end;

end.
```

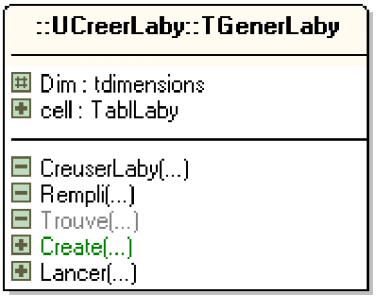




Aspect général de l'IHM de test du Labyrinthe :

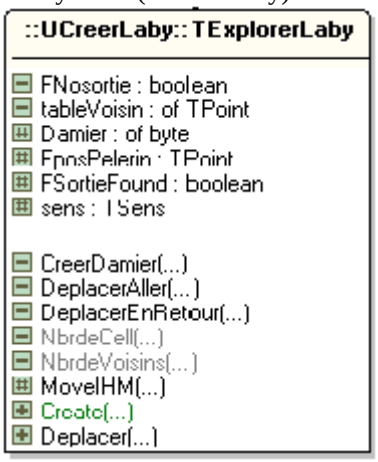



```
unit UCreerLaby;
{ un design pattern sur les labyrinthes }
interface
 uses Windows,Classes, Sysutils,graphics,extctrls,Controls,Dialogs;
 const
  maxTab=40;
  depart=0;
  carrefour=1;
  libre=2;
  markAller=3;
  markRetour=4;
  mur=10;
  sortie=20;
 type
  tdimensions=
  record
   largeur,hauteur:integer;
  end;
  TablLaby=array[0..maxTab,0..maxTab] of boolean;
  TSens=(aller,retour);


TGenerLaby=class
```



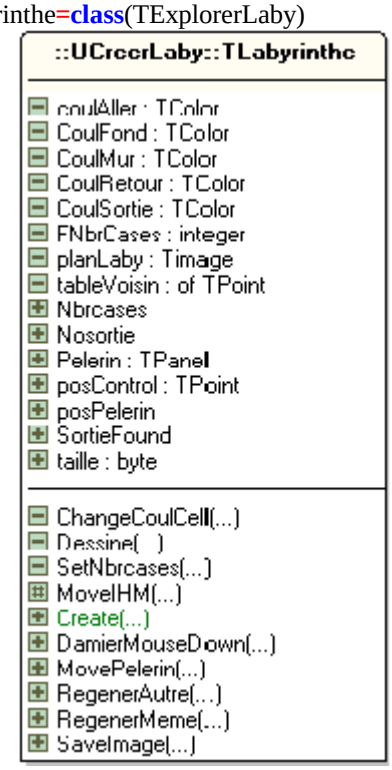

```
 private
  procedure Rempli;
  function Trouve(x,y:integer;dir:integer;var surf:TDimensions;var newDir:integer):boolean;
  procedure CreuserLaby(x,y,d:integer);
 protected
  Dim:TDimensions;
 public
  cell:TablLaby;
  procedure Lancer;
  constructor Create(Dim:TDimensions);virtual;
end;
```

TExplorerLaby=**class**(TGenerLaby)




```
 private
  tableVoisin:array[1..4]of TPoint;
  procedure CreerDamier;
  function NbrdeCell(etat:integer):integer;
  function NbrdeVoisins:integer;
  procedure DeplacerAller;
  procedure DeplacerEnRetour;
 protected
  sens:TSens;
  FposPelerin:TPoint;
  FSortieFound,FNosortie:boolean;
  Damier:array[0..maxTab,0..maxTab] of byte;
  procedure MoveIHM(x,y:integer;posExpl:TPoint;caillou:integer);virtual;
 public
  procedure Deplacer;
  constructor Create(Dim:TDimensions);override;
end;
```

TLabyrinthe=**class**(TExplorerLaby)




```
 private
  planLaby:Timage;
  tableVoisin:array[1..4]of TPoint;
  FNbrCases:integer;
  CoulMur,CoulFond,coulAller,CoulRetour,CoulSortie:TColor;
  procedure SetNbrcases(x:integer);
  procedure Dessine;
  procedure ChangeCoulCell(coord:TPoint;coul:TColor);
 protected
  procedure MoveIHM(x,y:integer;posExpl:TPoint;caillou:integer);override;
 public
  Pelerin:TPanel;
  taille:byte;
  posControl:TPoint;
  property Nbrcases:integer read FNbrcases write SetNbrcases;
  property SortieFound:boolean read FSortieFound;
  property Nosortie:boolean read FNosortie;
  property posPelerin:TPoint read FposPelerin write FposPelerin;
  procedure DamierMouseDown(Sender: TObject; Button: TMouseButton;
   shift: tshiftstate;
   x, y: integer);
  procedure MovePelerin(vers:TPoint);
  procedure RegenerMeme;
  procedure RegenerAutre;
  procedure SaveImage;
  constructor Create(Visu:TImage);
end ;

implementation

 { TLabyrinthe
   code Damier
     0 --> départ
     1 --> carrefour
     2 --> libre
```

```
   3 --> aller
   4 --> retour
  10 --> mur
  20 --> sortie

  La fonction récursive de creusement tirée de
  "logiciel pour créer des labyrinthes" de Raphaël Jolivet
  Mail: raphael.jolivet@wanadoo.fr
}


{-------------------------- TGenerLaby ---------------------- }

constructor TGenerLaby.Create(Dim:TDimensions);
begin
 inherited Create;
 self.Dim:=Dim;
 Lancer
end;

procedure TGenerLaby.Lancer;
begin
 Rempli;
 CreuserLaby(0,0,2);
end;

procedure TGenerLaby.CreuserLaby(x, y, d: integer);
var surface:TDimensions;
 dir:integer;
 begin
  dir:=random(1000)+1;
  while trouve(x,y,d,surface,dir)do
  begin
   cell[surface.largeur,surface.hauteur]:=true;
   cell[round((surface.largeur+x)/2),round((surface.hauteur+y)/2)]:=true;
   CreuserLaby(surface.largeur,surface.hauteur,dir);
  end;
 end;

procedure TGenerLaby.Rempli;
var i,j:integer;
begin
 for i:=0 to Dim.largeur do
  for j:=0 to Dim.hauteur do
   cell[i,j]:=false;
end;

function TGenerLaby.Trouve(x, y, dir: integer; var surf: TDimensions; var newDir: integer): boolean;
var surftemp:array[1..4] of TDimensions;
 d:array[1..4] of integer;
 i,k,h:integer;
 p:TDimensions;
 begin
  k:=0;
  for i:=0 to 3 do
  begin
   p.largeur:=x+2*round(cos(Pi/2*(i+dir) ) );
   p.hauteur:=y+2*round(sin(Pi/2*(i+dir) ) );
   if (p.largeur>=0) and (p.hauteur>=0) and (p.largeur<=Dim.largeur)
    and (p.hauteur<=Dim.hauteur)  then
```

```
  if not(cell[p.largeur,p.hauteur]) then
  begin
   k:=k+1;
   surftemp[k]:=p;
   d[k]:=i+dir;
  end;
 end;
 if k>0 then
 begin
  h:=1+random(k);
  surf:=surftemp[h];
  newDir:=d[h];
  result:=true;
 end
 else
  result:=false;
end;

 { ----------------------- TExplorerLaby ------------------------}
constructor TExplorerLaby.Create(Dim:TDimensions);
begin
 inherited;
 CreerDamier;
end;

procedure TExplorerLaby.CreerDamier;
var i,j:integer;
begin
 for i:=0 to Dim.largeur do
  for j:=0 to Dim.hauteur do
   if cell[i,j] then
    Damier[i,j]:=libre
   else
    Damier[i,j]:=mur ;
 Damier[0,0]:=depart;// point de départ
end;

procedure TExplorerLaby.Deplacer;
begin
 if not FSortieFound then
  if sens=aller then
   DeplacerAller
  else
   DeplacerEnRetour
 else
  MessageDlg('La sortie a déjà été trouvée !', mtWarning,[mbOk], 0);
end;

procedure TExplorerLaby.MoveIHM(x,y:integer;posExpl:TPoint;caillou:integer);
begin
 Damier[x,y]:= caillou;
end;

procedure TExplorerLaby.DeplacerAller;
var
 compteVoisins,ptr,i,j,NbrSortie:integer;
 begin
  i:=FposPelerin.x;
  j:=FposPelerin.y;
  NbrSortie:=NbrdeCell(sortie);
```

```
 if NbrSortie>0 then //c'est une sortie
 begin
  FposPelerin:=tableVoisin[1];
  Beep;
  FSortieFound:=true;
  MessageDlg('J"ai trouvé une sortie dans ce labyrinthe !', mtInformation ,[mbOk], 0);
  MoveIHM(i,j,FposPelerin,sortie);
  exit
 end
 else // ce n'est pas une sortie
 begin
  compteVoisins:=NbrdeVoisins;
  case
   comptevoisins of
   0:
   begin // cul-de-sac
    sens:=retour;
    MoveIHM(i,j,Point(i,j),markRetour);
   end;
   1:
   begin // couloir
    FposPelerin:=tableVoisin[1];
    if Damier[i,j]<>depart then
     MoveIHM(i,j,FposPelerin,markAller) //ce n'est pas l'entrée
    else // c'est l'entrée
     MoveIHM(i,j,FposPelerin,depart);
   end;
   2,3:
   begin // carrefour
    ptr:=random(compteVoisins)+1;
    FposPelerin:=tableVoisin[ptr];
    if Damier[i,j]<>depart then //ce n'est pas l'entrée
     MoveIHM(i,j,FposPelerin,carrefour)
    else // c'est l'entrée
     MoveIHM(i,j,FposPelerin,depart);
   end;
  end;
 end
end;

procedure TExplorerLaby.DeplacerEnRetour;
var
 NbrDepart,NewChemin,NbrCarrefour,NbrRetourPoss,i,j:integer;
 begin
  i:=FposPelerin.x;
  j:=FposPelerin.y;
  if (Damier[i,j]<>carrefour)and(Damier[i,j]<>depart) then
  begin
   NbrDepart:=NbrdeCell(depart);
   NbrRetourPoss:=NbrdeCell(markAller);
   NbrdeCell(carrefour);
   FposPelerin:=tableVoisin[1];
   if (NbrRetourPoss=0)and(NbrDepart=1) then //on va au départ
   begin
    NbrdeCell(depart);
    FposPelerin:=tableVoisin[1];
   end;
   MoveIHM(i,j,FposPelerin,markRetour);
  end
  else // on est revenu à un carrefour (point de départ inclu)
```

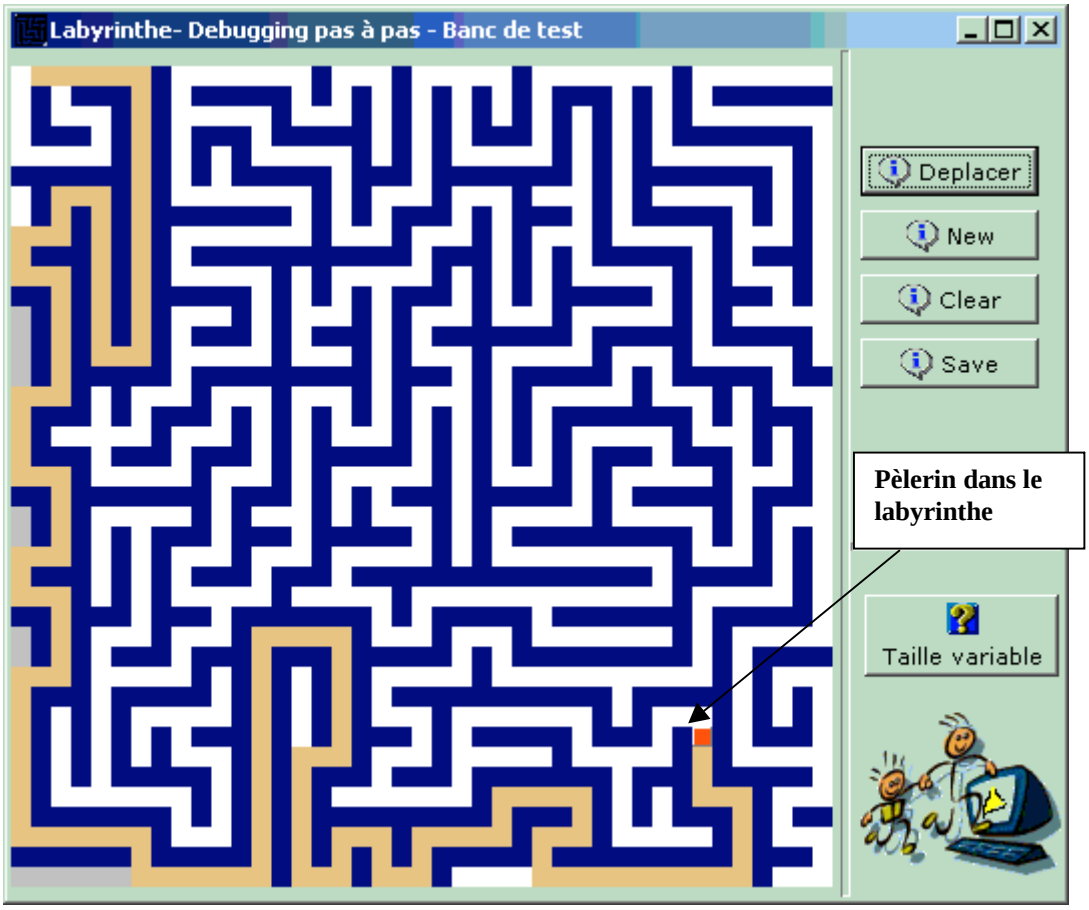
```
  begin
   //-- on est sur un vrai carrefour
   NewChemin:=NbrdeCell(libre);
   if NewChemin>0 then //au moins un chemin encore non exploré
   begin
    if Damier[i,j]<>depart then
     MoveIHM(i,j,FposPelerin,markAller)
    else
     MoveIHM(i,j,FposPelerin,depart);
    sens:=aller;
   end
   else // plus de chemin à explorer
   begin
    NbrRetourPoss:=NbrdeCell(markAller); // il n'y en a qu'une
    {-- tableVoisin[1] contient les coordonnées  de cette cellule (de genre markaller)      }
    if (NbrRetourPoss=0)and(Damier[i,j]=depart) then //on est sur la case départ
    begin
     FNosortie:=true;
     MessageDlg('Il n''y a pas de sortie dans ce labyrinthe !', mtWarning,[mbOk], 0);
     exit
    end;
    FposPelerin:=tableVoisin[1];
    MoveIHM(i,j,FposPelerin,markRetour);
   end
  end;
 end;

function TExplorerLaby.NbrdeCell(etat: integer): integer;
var compte:integer;
begin
 compte:=0;
 if FposPelerin.y>0 then
  if Damier[FposPelerin.x,FposPelerin.y-1]=etat then
  begin
   compte:=compte+1;
   tableVoisin[compte]:=Point(FposPelerin.x,FposPelerin.y-1);
  end;
 if FposPelerin.x<Dim.largeur  then
  if Damier[FposPelerin.x+1,FposPelerin.y]=etat then
  begin
   compte:=compte+1;
   tableVoisin[compte]:=Point(FposPelerin.x+1,FposPelerin.y);
  end;
 if FposPelerin.y<Dim.hauteur  then
  if Damier[FposPelerin.x,FposPelerin.y+1]=etat then
  begin
   compte:=compte+1;
   tableVoisin[compte]:=Point(FposPelerin.x,FposPelerin.y+1);
  end;
 if FposPelerin.x>0 then
  if Damier[FposPelerin.x-1,FposPelerin.y]=etat then
  begin
   compte:=compte+1;
   tableVoisin[compte]:=Point(FposPelerin.x-1,FposPelerin.y);
  end;
 result:=compte
end;

function TExplorerLaby.NbrdeVoisins: integer;
begin
```

```
 result:=NbrdeCell(libre)
end;

{ ------------------------  TLabyrinthe --------------------------}
constructor TLabyrinthe.Create(Visu:TImage);
begin
 if not Assigned(Visu) then
 begin
  MessageDlg('Il faut utiliser un TImage pour afficher le plan !', mtError ,[mbOk], 0);
  Exit;
 end;
 planLaby:=Visu;
 FNbrCases:=maxTab;
 Dim.largeur:=FNbrCases;
 Dim.hauteur:=FNbrCases;
 inherited Create(Dim);
 taille:=10;
 sens:=aller;
 FSortieFound:=false;
 FNosortie:=false;
 CoulMur:=clNavy;
 CoulFond:=clWhite;
 coulAller:=$00ABCEFE;
 CoulRetour:=clsilver;//$00E1FEFF;
 CoulSortie:=clFuchsia;
 Randomize;
 planLaby.width:=(Dim.largeur+1)*taille;
 planLaby.height:=(Dim.hauteur+1)*taille;
 planLaby.OnMouseDown:=DamierMouseDown;
 posControl:=Point(planLaby.left,planLaby.top);
 Dessine;
end;

procedure TLabyrinthe.DamierMouseDown(Sender: TObject;  button: tmousebutton;
 shift: tshiftstate;   x, y: integer);
var i,j:integer;
begin
 i:=x div taille;
 j:=y div taille;
 if damier[j,i]=mur then
  exit;
 Damier[j,i]:=sortie;
 TImage(Sender).canvas.brush.color:=CoulSortie;
 TImage(Sender).canvas.fillrect(rect(i*taille,j*taille,(i+1)*taille,(j+1)*taille));
end;

procedure TLabyrinthe.Dessine;
var i,j,Larg,Long:integer;
begin
 for i:=0 to Dim.largeur do
  for j:=0 to Dim.hauteur do
  begin
   if cell[j,i] then
   begin
    planLaby.canvas.brush.color:=CoulFond;
   end
   else
   begin
    planLaby.canvas.brush.color:=CoulMur
   end;
```

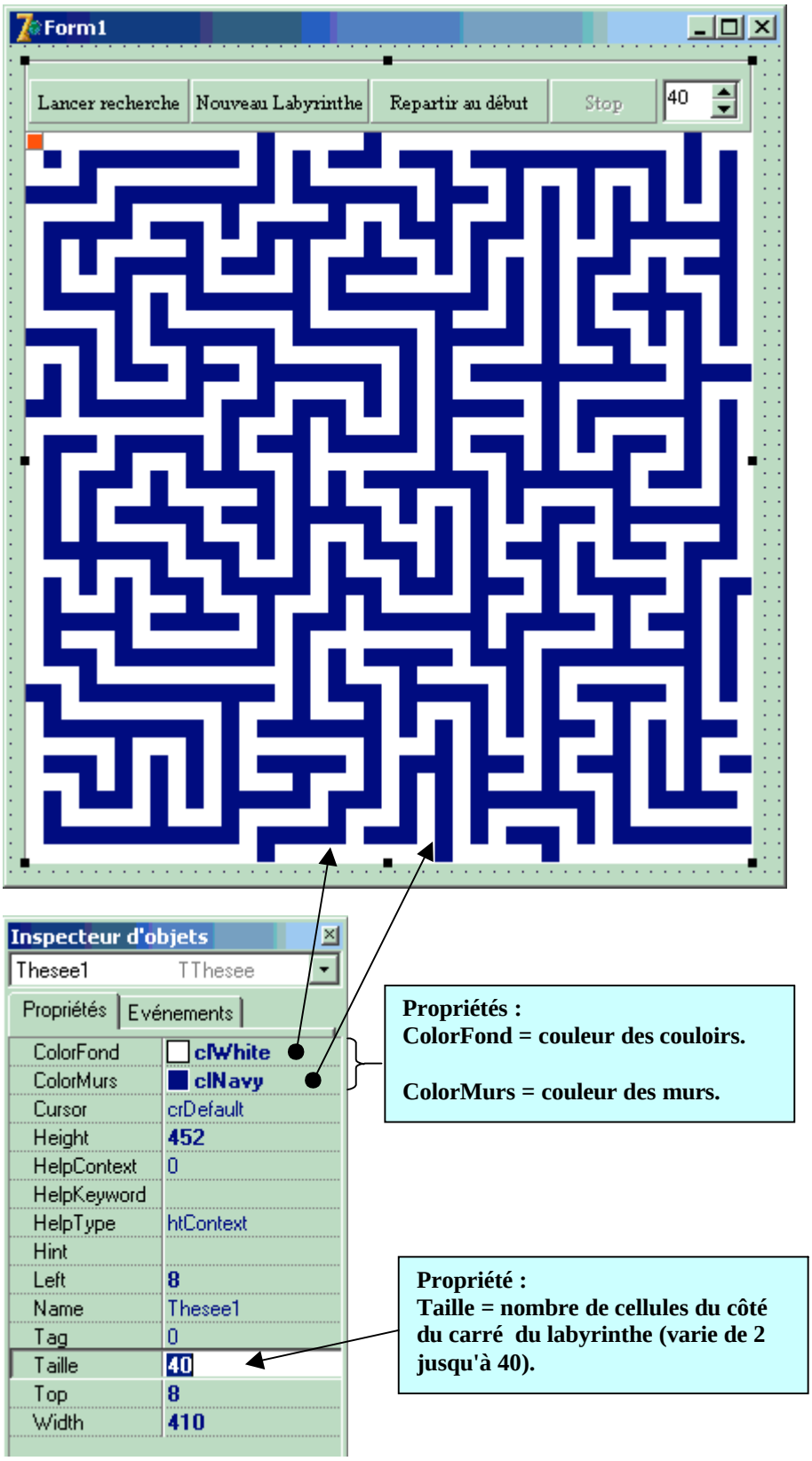
```
   planLaby.canvas.fillrect(rect(i*taille,j*taille,(i+1)*taille,(j+1)*taille));
  end ;
 if Assigned(planLaby) then
 begin
  Larg:= (Dim.largeur+1)*taille+1;
  Long:= (Dim.hauteur+1)*taille+1;
  planLaby.canvas.Pen.Color:=clBlack;
  planLaby.canvas.Pen.Width:=2;
  planLaby.canvas.MoveTo(0,Larg);
  planLaby.canvas.LineTo(Long,Larg);
  planLaby.canvas.LineTo(Long,0);
 end
end;

procedure TLabyrinthe.RegenerAutre;
begin
 Lancer;
 RegenerMeme;
end;

procedure TLabyrinthe.RegenerMeme;
begin
 planLaby.canvas.brush.color:=clSilver;
 planLaby.canvas.fillrect(rect(0,0,planLaby.width,planLaby.height));
 CreerDamier;
 Dessine;
 posPelerin:=Point(0,0);
 MovePelerin(Point(0,0));
 sens:=aller;
 FSortieFound:=false;
 FNosortie:=false;
end;

procedure TLabyrinthe.SaveImage;
var InputString:string;
begin
 InputString:= InputBox('Nom du fichier', 'ENTREZ', 'LabyPerso.bmp');
 planLaby.Picture.SaveToFile(InputString);
end;

procedure TLabyrinthe.SetNbrcases(x: integer);
begin
 if x in [2..maxTab] then
 begin
  FNbrcases:=x;
  Dim.largeur:=x;
  Dim.hauteur:=x;
  RegenerAutre;
 end
end;
//----------- Le déplacement dans le labyrinthe
procedure TLabyrinthe.MovePelerin(vers:TPoint);
begin
 Pelerin.Top:=posControl.y+vers.y;
 Pelerin.Left:=posControl.x+vers.x;
end;

procedure TLabyrinthe.ChangeCoulCell(coord:TPoint;coul:TColor);
begin
 planLaby.canvas.brush.color:=coul;
```

```
 planLaby.canvas.fillrect(rect(coord.y*taille,coord.x*taille,
 (coord.y+1)*taille,(coord.x+1)*taille));
 planLaby.Update
end;

procedure TLabyrinthe.MoveIHM(x, y: integer; posExpl: TPoint;
 caillou: integer);
 begin
  inherited;
  MovePelerin(Point((posExpl.y)*taille, (posExpl.x)*taille));
  case
   caillou of
   markaller:
   begin
    ChangeCoulCell(Point(x,y),CoulAller);
    Pelerin.Color:=clRed;
   end;
   markretour:
   begin
    ChangeCoulCell(Point(x,y),CoulRetour);
    Pelerin.Color:=clBlue;
   end;
   carrefour,sortie:ChangeCoulCell(Point(x,y),CoulAller);
  end;
 end;

end.
```

Aspect après plusieurs appuis sur le bouton Déplacer :

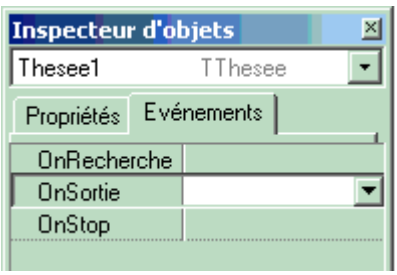


Extension de la solution à une nouvelle version sous forme de composant d'animation réutilisable à déposer dans la palette des composants Delphi dans l'onglet 'Exemples', il est construit par association sur un TPanel du Labyrinthe proprement dit dessiné sur un TImage, de 4 TSpeedButton permettant d'intervenir sur la génération d'un nouveau labyrinthe et sur l'animation de la recherche dans le labyrinthe et enfin d'unTSpinEdit qui permet de changer la taille du labyrinthe (valeur entre 2 et 40), l'animation est assurée par un TTimer :



Le composant est sensible à 3 événements :

OnRecherche : qui se produit dès que le pèlerin commence sa recherche de sortie dans le labyrinthe.
OnSortie : qui se déclenche dès que le pèlerin a trouvé une sortie dans le labyrinthe.
OnStop : qui se déclenche dès que le pèlerin s'est arrêté dans l'un des 3 cas suivants : on arrête l'animation avec le bouton stop, le pèlerin s'est rendu compte qu'il n'y avait pas de sortie dans le labyrinthe, le pèlerin a trouvé une sortie dans le labyrinthe.



Dès que l'on clique sur le bouton lancer recherche, l'animation commence, le pèlerin se déplace dans le labyrinthe.

Ce composant utilise la **unit** UCreerLaby**;** précédente qui contient les 3 classes de base du labyrinthe :

```
unit ULabyrinthe;
{ un composant obtenu à partir du   design pattern sur les labyrinthes }
interface

 uses UCreerLaby,Messages,Classes,extctrls,Controls, graphics,buttons,
  Spin;
  {
   code Damier
     0 --> départ
     1 --> carrefour
     2 --> libre
     3 --> aller
     4 --> retour
    10 --> mur
    20 --> sortie
  }
 type
  TThesee =class(TCustomPanel)
 private
  Ftaille:integer;
  Fstop:boolean;
  FOnStop,FOnRecherche,FOnSortie:TNotifyEvent;
  TheseeExplor,Command:Tpanel;
  Plan:Timage;
  LabyMinos:TLabyrinthe;
  Tempo:TTimer;
  SpeedBtnLancer,SpeedBtnNew,SpeedBtnClear,SpeedBtnStop:TSpeedButton;
  SpinEditCases: TSpinEdit;
  procedure Timing(Sender: TObject);
  procedure SpeedBtnLancerClick(Sender: TObject);
  procedure SpeedBtnNewClick(Sender: TObject);
  procedure SpeedBtnClearClick(Sender: TObject);
  procedure SpeedBtnStopClick(Sender: TObject);
  procedure SpinChange(Sender: TObject);
  procedure SetTaille(x:integer);
  procedure WMSizeRedim(var Msg:TWMsize); message WM_size;
  function GetColorMurs:TColor;
  function GetColorFond:TColor;
```

```
  procedure SetColorMurs(x:TColor);
  procedure SetColorFond(x:TColor);
 public
  constructor Create(AOwner: TComponent); override;
 published
  property Taille:integer read Ftaille write SetTaille;
  property ColorMurs:TColor read GetColorMurs write SetColorMurs;
  property ColorFond:TColor read GetColorFond write SetColorFond;
  property OnStop:TNotifyEvent read FOnStop write FOnStop;
  property OnSortie:TNotifyEvent read FOnSortie write FOnSortie;
  property OnRecherche:TNotifyEvent read FOnRecherche write FOnRecherche;
end;

  procedure Register;

implementation

 procedure Register;
 begin
  RegisterComponents( 'Exemples', [TThesee] );
 end;
 {---------------------- TThesee -----------------------}
 constructor TThesee.Create(AOwner: TComponent);
 var i:integer;
 begin
  inherited;
  self.Width:=410;
  self.Height:=452;
  self.BevelInner:=bvLowered;
  self.BevelOuter:=bvNone;
  self.Font.Name:='Times New Roman';
  self.Font.Size:=8;
  self.parent:=TWinControl(AOwner);
  Command:=Tpanel.Create(self);
  Command.parent:=self;
  Command.SetBounds(0,0,410,40);
  Command.Align:=alTop;
  SpeedBtnLancer:=TSpeedButton.Create(Command);
  SpeedBtnLancer.SetBounds(2,10,90,25);
  SpeedBtnLancer.Caption:='Lancer recherche';
  SpeedBtnNew:=TSpeedButton.Create(Command);
  SpeedBtnNew.SetBounds(93,10,100,25);
  SpeedBtnNew.Caption:='Nouveau Labyrinthe';
  SpeedBtnClear:=TSpeedButton.Create(Command);
  SpeedBtnClear.SetBounds(194,10,100,25);
  SpeedBtnClear.Caption:='Repartir au début';
  SpeedBtnStop:=TSpeedButton.Create(Command);
  SpeedBtnStop.SetBounds(295,10,60,25);
  SpeedBtnStop.Caption:='Stop';
  SpeedBtnStop.Enabled:=false;
  for i:=0 to Command.ComponentCount-1 do
   if Command.Components[i] is TSpeedButton then
    with (Command.Components[i] as TSpeedButton)do
    begin
     parent:=Command;
     flat:=true;
     cursor:=crHandPoint
    end;
  SpeedBtnLancer.OnClick:=SpeedBtnLancerClick;
  SpeedBtnNew.OnClick:=SpeedBtnNewClick;
```

```
 SpeedBtnClear.OnClick:=SpeedBtnClearClick;
 SpeedBtnStop.OnClick:=SpeedBtnStopClick;
 SpinEditCases:=TSpinEdit.Create(Command);
 SpinEditCases.parent:=Command;
 SpinEditCases.SetBounds(358,10,45,25);
 SpinEditCases.MaxValue:=maxTab;
 SpinEditCases.MinValue:=2;
 SpinEditCases.Value:=maxTab;
 SpinEditCases.Increment:=2;
 SpinEditCases.OnChange:= SpinChange;
 Plan:=Timage.Create(self);
 Plan.parent:=self;
 Plan.AutoSize:=false;
 Plan.SetBounds(0,42,410,410);
 Plan.canvas.brush.color:=clSilver;
 Plan.Align:=alClient;
 Plan.canvas.fillrect(rect(0,0,width,height));
 Ftaille:=maxTab;
 Fstop:=true;
 TheseeExplor:=Tpanel.Create(self);
 TheseeExplor.parent:=self;
 TheseeExplor.Color:=clRed;
 LabyMinos:=TLabyrinthe.Create(Plan);
 LabyMinos.Pelerin:=TheseeExplor;
 TheseeExplor.Width:= LabyMinos.taille;
 TheseeExplor.Height:= LabyMinos.taille;
 LabyMinos.posPelerin:=Point(0,0);
 LabyMinos.MovePelerin(Point(0,0));
 Tempo:=TTimer.Create(self);
 Tempo.Interval:=50;
 Tempo.Enabled:=false;
 Tempo.OnTimer:=Timing;
end;

procedure TThesee.Timing(Sender: TObject);
begin
 if not LabyMinos.sortieFound and not LabyMinos.Nosortie
  and not fstop then
      labyminos.deplacer
 else
 begin
  tempo.Enabled:=false;
  if assigned(FOnStop) then
      FOnStop (self);
  if LabyMinos.sortieFound then
  begin
   if assigned(FOnSortie) then
        FOnSortie (self);
   SpeedBtnStop.Enabled:=false;
   SpeedBtnLancer.Enabled:=false;
  end;
  if LabyMinos.Nosortie then
  begin
   SpeedBtnStop.Enabled:=false;
   SpeedBtnLancer.Enabled:=false;
  end
 end
end;

procedure TThesee.SpeedBtnLancerClick(Sender: TObject);
```

Gestionnaire de l'événement Ontimer de l'objet Tempo de type TTimer :

Toutes les 50 ms le pèlerin se déplace d'une cellule dans le labyrinthe.

Gestionnaire de l'événement Ontimer de l'objet Tempo de type TTimer :

Si une sortie a été trouvée, ou il n'y a pas de sortie, ou on a demandé l'arrêt de l'animation, le timer se désactive et tout est stoppé.

```
begin
 Fstop:=false;
 Tempo.Enabled:=true;
 SpeedBtnStop.Enabled:=true;
 SpeedBtnStop.Caption:='Stop';
 if assigned(FOnRecherche) then
    FOnRecherche (self)
end;

procedure TThesee.SpeedBtnNewClick(Sender: TObject);
begin
 LabyMinos.RegenerAutre;
 SpeedBtnLancer.Enabled:=true;
end;

procedure TThesee.SpeedBtnClearClick(Sender: TObject);
begin
 LabyMinos.RegenerMeme;
 SpeedBtnLancer.Enabled:=true;
end;

procedure TThesee.SpeedBtnStopClick(Sender: TObject);
begin
 if SpeedBtnStop.Caption='Stop' then
 begin
  Fstop:=true;
  SpeedBtnStop.Caption:='Continuer';
 end
 else
 begin
  Fstop:=False;
  SpeedBtnStop.Caption:='Stop';
  Tempo.Enabled:=true;
  if assigned(FOnRecherche) then
         FOnRecherche (self)
 end
end;

procedure TThesee.SpinChange(Sender: TObject);
begin
 try
  if not odd(SpinEditCases.Value) then
   if SpinEditCases.Value in [2..maxTab] then
    LabyMinos.Nbrcases:= SpinEditCases.Value
 except
  LabyMinos.Nbrcases:=2
 end
end;

procedure TThesee.SetTaille(x: integer);
begin
 if x in [2..maxTab] then
 begin
  if odd(x) then
   x:=x+1;
  LabyMinos.Nbrcases:=x;
  Ftaille:=x;
  SpinEditCases.Value:=x;
 end
end;
```

```
procedure TThesee.WMSizeRedim(var Msg: TWMsize);
begin
 self.Width:=410;
 self.Height:=452;
end;

function TThesee.GetColorFond: TColor;
begin
 result:=LabyMinos.CoulFond
end;

function TThesee.GetColorMurs: TColor;
begin
 result:=LabyMinos.CoulMur
end;

procedure TThesee.SetColorFond(x: TColor);
begin
 LabyMinos.CoulFond :=x;
 LabyMinos.RegenerMeme;
end;

procedure TThesee.SetColorMurs(x: TColor);
begin
 LabyMinos.CoulMur:=x;
 LabyMinos.RegenerMeme;
end;

end.
```

Remerciements

à Mrs : (*pour leur patiente relecture et leurs conseils précieux et pertinents pour la première version de la partie cours du package contenue dans les chapitres 1 à 8 du livre)*

M.Lai consultant expert en technologie objets à ETAC et Professeur d'informatique associé, à l'IUP d'Aix-Marseille auteur lui-même d'ouvrages solides et de niveau spécialisé sur la conception orientée objet avec Java.

A.Rouillon Maître de conférence Hors-classe en informatique retr., ancien directeur de l'ecole d'ingénieur de Tours et ancien directeur de l'E3i.

Remerciements à Mrs : (*pour les corrections d'erreurs et conseils*)
G.Bougeret, *C.Gervais* (en particulier pour le chapitre 9 du cours Java version initiale).

J-M Dumas, professeur d'informatique, spécialiste SGBD de notre filière d'enseignement qui m'a fourni de précieuses informations de terrain sur les BD ainsi que des remarques pédagogiques sur le chapitre 7.5 et son épouse *Nicole* pour ses corrections du même chapitre.

A tous les rares internautes qui ont bien voulu signaler une erreur dans la rédaction.

Remerciements : (*pour participation*)
P. di Scala professeur de mathématiques au lycée G.Pompidou de Mauriac pour la rédaction des exercices et projets Delphi sur les bases de données du chapitre 7 et dans le CD-ROM.

A mon épouse Dominique pour son soutien et sa patience qui me permettent de consacrer de nombreuses heures à la construction du package et des cours inclus et surtout comme la seule personne en dehors de moi qui a eu la constance de relire entièrement toutes les pages de l'ouvrage, alors que l'informatique n'est pas sa tasse de thé.

Remerciements : (*diffusion de la connaissance*)

- A l'Université de Tours qui supporte et donne accès à la partie Internet du package pédagogique à partir de sa rubrique "cours en ligne".

- Au club des développeurs francophones qui héberge un site miroir du précédent et qui recommande le package pédagogique (http://rmdiscala.developpez.com/cours/) à ses visiteurs débutants.

Remerciements : (*Anticipés*)

A tous ceux qui voyant des erreurs dans l'actuel document auraient l'obligeance de me les signaler pour une future mise à jour, e-mail : discala@univ-tours.fr .

Annexe

Syntaxes de base comparées LDFA - Delphi - Java/C#

LDFA	Delphi	Java - C#
Ω (instruction vide)	pas de traduction	pas de traduction
debut i1 ; i2; i3; ..... ; ik **fin**	**begin** i1 ; i2; i3; ..... ; ik **end**	**{** i1; i2; i3; ..... ; ik; **}**
x <-– a	x := a	x **=** a **;**
;	(ordre d'exécution) ;	(ordre d'exécution + fin instruction) **;**
Si P **alors** E1 **sinon** E2 **Fsi**	**if** P **then** E1 **else** E2 *(attention défaut, pas de fermeture !)*	**if (P)** E1**; else** E2 **;** *(attention défaut, pas de fermeture !)*
Tantque P **faire** E **Ftant**	**while** P **do** E *(attention, pas de fermeture)*	**while (P)** E **;** *(attention, pas de fermeture)*
répeter E **jusquà** P	**repeat** E **until** P	**do** E**; while (!P)**
lire (x1,x2,x3.....,xn)	read(fichier,x1,x2,x3.....,xn) readln(x1,x2,x3.....,xn) Get(fichier)	en C# : **System.Console.Read( )** **System.Console.ReadLine( )** en Java : **System.in.read( )** **System.in.readln( )**
ecrire (x1,x2,x3.....,xn)	write(fichier,x1,x2,x3.....,xn) writeln(x1,x2,x3.....,xn) Put(fichier)	en C# : **System.Console.Write( )** **System.Console.WriteLine( )** en Java : **System.out.print( )** **System.out.println( )**
pour x<-a **jusquà** b **faire** E **Fpour**	**for** x:=a **to** b **do** E *(croissant)* **for** x:=a **downto** b **do** E *(décroissant)* *(attention, pas de fermeture)*	**for (x = a ; x < b ; x++) E ;** **for (x = a ; x < b ; x--) E ;** *(attention, pas de fermeture)*
SortirSi P	**if** P **then** Break *(sort d'une boucle)* **if** P **then** exit *(sort d'une méthode)*	**if (P)** Break **;** *(sort d'une boucle)* **if (P)** return **;** *(sort d'une méthode)*
N (entiers naturels)	**integer**	**uint** en C# **int** en Java

Z (entiers relatifs)	**integer**	**int** en C# **int** en Java
Q (rationnels)	**real**	**double** en C# et Java
R (réels)	**real**	**double** en C# et Java
{ Vrai , Faux } (logique)	**boolean**	**Boolean** en C# **bool** en Java
caractère	**char**	**char** en C# et Java
+ , - , / , *	**+ , - , / , ***	**+ , - , / , ***
> , < , = , ≠	**> , < , = ,<>**	**> , < , = , !=**
≥ , ≤	>= , <=	**>= , <=**
¬, ∧ , ∨	**not , and , or**	**! , && , ||**

Bibliographie

Chapitre 1

Lorrains, Réseaux téléinformatiques, Hachette technique, Paris (1979).
P. Franken, OS/2 2.0, Micro Application, Paris (1992).
Data Becker, Le meilleur de Windows 95, Micro Application, Paris (1995).
H. Boucher, Architecture de l'ordinateur – Tome 3 – Logiciel, Cepadues Editions, Toulouse (1980).
Victor Sandoval, Intranet, le réseau d'entreprise, Hermes, Paris (1996).
Pierre-Alain Goupille, Technologie des ordinateurs et des réseaux, Masson, Paris (1996).
Daniel Etiemble, Architecture des processeurs Risc, Armand Colin, Paris (1991).
Y. Nishinuma, R. Espesser, Unix premiers contacts, Eyrolles, Paris (1986).
S. Krakowiak, Principes des systèmes d'exploitation des ordinateurs, Dunod, Paris (1985).
Guy Pujolle, La télématique réseaux et applications, Eyrolles, Paris (1983).
Guy Pujolle, Les réseaux d'entreprise, Eyrolles, Paris (1983).
Cornafion, Systèmes informatiques répartis, Dunod, Paris (1981).
Crocus, Systèmes d'exploitation des ordinateurs, Dunod, Paris (1975).
Philippe Dax, CP/M et sa famille, Eyrolles, Paris (1982).
Daniel-Jean David, Les systèmes à microprocesseurs, éditests, Paris (1982).
Alain Pinaud, Programmer en assembleur, P.S.I., Lagny (1982).
Data Becker, Le grand livre MS-DOS 5.0, Micro Application, Paris (1991).
A. Villard, M. Miaux, Un microprocesseur pas à pas, ETSF, Paris (1983).
Wladimir Mercouroff, Les ordinateurs, Cedic/Fernand Nathan, Paris (1980).
Adam Osborne, Initiation aux micro-ordinateurs, éditions Radio, Paris (1981).
Frédéric Hoste, Les réseaux locaux d'entreprise, édi test, Paris (1983).
C. Macchi, J.-F. Guilbert, Téléinformatique, Dunod, Paris (1979).
H. Lilen, Du microprocesseur au micro-ordinateur, éditions Radio, Paris (1980).
J. du Roscoät, Principes et problèmes d'un système d'exploitation d'ordinateur, Masson et Cie, Paris (1972).
C. Carrez, Les systèmes informatiques, Dunod, Paris (1990).
Guy Lacroix, Le mirage internet – Enjeux économiques et sociaux, Vigot, Paris (1997).
A. Tanenbaum, Les systèmes d'exploitation, InterEditions, Paris (1989).
A. Tanenbaum, Architecture de l'ordinateur, InterEditions, Paris (1987).
Donald H. Sanders, L'univers des ordinateurs, McGraw Hill, Paris (1984).
J.Steiner, Comprendre choisir et utiliser les microprocesseurs, Osman-Eyrolles, Paris (2000)
E.Charton, créer un intranet, campus press, Paris (2000)
O.Pavie, les réseaux, campus press, Paris (2000)
J.Casad, B.Willsey, TCP/IP , campus press, Paris (1999)
T.Drilling, guide de l'utilisateur LINUX, Micro Application, Paris (2000)
A.Arnold, I.Guerssarian, mathématiques pour l'informatique, Masson, Paris (1997).
N.Wielsch & al, Kit de démarrage Linux Mandrake 7.2, Micro Application, Paris (2000)
G.Pujolle Best of : les réseaux , Eyrolles, Paris (2002)
A.Tanenaum, Systèmes d'exploitation, Pearson education 2° éd.,Paris (2003)
Th.Lucas et al Initiation à la logique formelle, De Boeck université, Bruxelles (2003)
P.Maurette, Programmez en assembleur, Micro Application, Paris (2004)

Chapitre 2

P. Lignelet, PASCAL manuel de base – Tome 1, Masson, Paris (1980).
P. Lignelet, PASCAL techniques de programmation - Tome 2, Masson, Paris (1980).
D.-J. David, J.-L. Deschamps, Programmer en Pascal avec version Turbo Pascal, P.S.I., Lagny (1985).
Miller, Programmer en Pascal pour scientifiques et ingénieurs, Sybex, Paris (1982).
J.-L. Nebut, Théorie et pratique du langage Pascal, éditions Technip, Paris (1980).
J. Lonchamp, Les langages de programmation, Masson, Paris (1989).
O. Lecarme, J.-L. Nebut, Pascal pour programmeurs, McGraw-Hill, Paris (1985).
Michel Marchand, Mathématique discrète – Outil pour l'informaticien, De Boeck Université, Bruxelles (1989).
André Arnold, Irène Guessarian, Mathématiques pour l'informatique, Masson, Paris (1997).

R. Faure, B. Lemaire, Mathématiques pour l'informaticien, Gauthier-Villars, Paris (1973).
A. Kaufmann, Mathématiques nouvelles pour mieux comprendre l'informatique, Entreprise Moderne d'Edition, Paris (1974).
M. Gross, A. Lentin, Grammaires formelles, Gauthier-Villars, Paris (1970).
K. Weiskamp, L. Heiny, B. Flamig, Borland C++ - Programmation orientée objet, Sybex, Paris (1991).
A.R. Feuer, Langages C - Problèmes et exercices, Masson, Paris (1991).
Data Becker, Turbo C version 2, Micro Application, Paris (1990).
Ch. Bonnin, Exercices pratiques de programmation en Cobol A.N.S. 74, Eyrolles, Paris (1982).
J. Guyot, C. Vial, Arbres, tables & algorithmes, Eyrolles, Paris (1992).
C. Berthet, Le langage de programmation PL/1, Dunod, Paris (1977).
Thomas Lachand-Robert, Introduction à Turbo C++, Sybex, Paris (1990).
M. Thorin, ADA – Manuel complet du langage avec exemples, Eyrolles, Paris (1981).
M. Gauthier, ADA – Un apprentissage, Dunod, Paris (1989).
J.-F. Macary, C. Nicolas, Programmation Java, Eyrolles, Paris (1996).
J. Laborde, Cours pratique de langage algol, Dunod, Paris (1967).
Bjarne Stroustrup, Le langage C++, InterEditions, Paris (1989).
Groupe Algol de l'AFCET, Définition du langage algorithmique Algol 68, Hermann, Paris (1972).
Jean-Michel Hufflen, Programmation fonctionnelle en schème, Masson, Paris (1996).
J.S. Chion, E.F. Clermann, Le langage Algol W, Presses Universitaires de Grenoble, Grenoble (1973).
H. Farreny, LISP, Masson, Paris (1984).
Tony Hasemer, LISP, InterEditions, Paris (1985).
C. Delannoy, Apprendre à programmer en Fortran, Eyrolles, Paris (1982).
D. Le Verrand, Le langage ADA, Dunod, Paris (1982).
K. Jensen, N. Wirth, Pascal – Manuel de l'utilisateur (rapport initial), Eyrolles, Paris (1981).
Pierre Weis, Xavier Leroy, Le langage Caml, InterEditions, Paris (1993).

H. Gallaire, Techniques de compilation, Cepadues édition, Toulouse (1984).
Nino Silverio, Réaliser un compilateur – Les outils Lex et Yacc, Eyrolles, Paris (1994).
A. Aho, R. Sethi, J. Ullman, Compilateurs – Principes, techniques et outils, InterEditions, Paris (1989).
Michel Hughes, Initiation mathématique aux grammaires formelles, Larousse, Paris (1972).
B. Groc, M. Bouhier, La programmation par syntaxe, Dunod, Paris (1990).
A. Aho, J. Ullman, Concepts fondamentaux de l'informatique, Dunod, Paris (1993).
Comprendre la linguistique, Marabout université, Verviers (1975).
Noam Chomsky, Structures syntaxiques, éditions du Seuil (1969).
N. Chomsky, G.A. Miller, L'analyse formelle des langues naturelles, Gauthier-Villars (1968).
J.-M. Autebert, Théorie des langages et des automates, Masson, Paris (1994).
D.Grune et al, cours et exercices compilateurs, Dunod, Paris (2002)

Chapitre 3 & 4

J. Arsac, La construction de programmes structurés, Dunod, Paris (1977).
Thomas Forse, Qualimétrie des systèmes complexes, Les Editions d'Organistaion, Paris (1989).
Grégoire, Informatique – Programmation 1, Masson, Paris (1986).
Grégoire, Informatique – Programmation 2, Masson, Paris (1988).
Grégoire, Informatique – Programmation 3, Masson, Paris (1990).
M. Thorin, Génie logiciel, Masson, Paris (1984).
M.C. Gaudel et al., précis de génie logiciel,Masson,Paris(1996)
J. Arsac, Premières leçons de programmation, Cedic/Fernand Nathan, Paris (1980).
G. Chaty, J. Vicard, L'algorithmique de la pratique à la théorie, Cedic/Fernand Nathan, Paris (1983).
F.H. Raymond, Programmation : Introduction théorique en vue de la pratique, Masson, Paris (1984).
C. froidevaux, M.C. Gaudel, M. Soria, Types de données et algorithmes, McGraw Hill, Paris (1990).
Ian Sommerville, Le génie logiciel et ses applications, InterEditions,Paris (1988).
P. Berlioux, Ph. Bizard, Algorithmique, Dunod, Paris (1983).
P.C. Scholl, J.-P. Peyrin, Schémas algorithmiques fondamentaux, Masson,Paris (1988).
M. Lucas, Algorithmique et représentation des données 2, Masson, Paris (1983).
Hua Thanh te, J.-F. dazy, D. Enselme, Programmation CNAM niveau A, Masson, Paris (1981).
Grégoire, Cours d'informatique – Programmation 1, ESI, Paris (1984).
Grégoire, Cours d'informatique – Programmation 2, ESI, Paris (1984).
P. Lignelet, Algorithmique – Niveau de base 1, Masson, Paris (1981).

P. Lignelet, Algorithmique – Niveau avancé 2, Masson, Paris (1981).
F.H. Raymond, Informatique – Programmation CNAM cours A, Masson, Paris (1980).
J. Lonchamp, Les langages de programmation, Masson, Paris (1989).
Patrick Jaulent, Génie logiciel : les méthodes, Armand Collin, Paris (1992).
Harold Abelson et Al, Structure et interprétation des programmes informatiques, InterEditions, paris (1989).
A. Ducrin, Programmation 2, Dunod, Paris (1984).
JP. Fournier, Passeport pour l'algorithmique objet, International Thomson Publishing France, paris (1997).
Bertrand Meyer, Introduction à la théorie des langages de programmation, InterEdition, Paris (1992).
M. Soberman, Génie logiciel en informatique de gestion, Eyrolles, Paris (1992).
C. Pair, R. Mohr, R. Schott, Construire les algorithmes, Dunod, Paris (1988).
J. Courtin, L. Kowarski, Initiation à l'algorithmique et aux structures de données, Dunod, Paris (1994).
R.C. Backhouse, Construction et vérification de programmes, Masson, Paris (1989).
J. Biondi, G. Clavel, Introduction à la programmation 1, Masson, Paris(1984).
P.C. Scholl, Algorithmique et représentation des données 3, Masson, Paris (1984).
Ian Sommerville, Le génie logiciel, Addison-Wesley, Paris (1992).
J.F. Monin, Comprendre les méthodes formelles, Masson, Paris (1996).
Leslie B. Wilson, Robert G. Clark, Langages de programmation comparés, Addison-Wesley, Paris (1993).
Jacques Arsac, Préceptes pour programmer, Dunod, Paris (1991).
B Liskov, J.Guttag, La maitrîse du développement de logiciel, Les Editions d'organisation, Paris (1990).
B. Ibrahim, C. Pellegrini, Structuration des données informatiques, Dunod, Paris (1989).
Aho Hopcroft Ullman, Structures de données et algorithmes, InterEditions, Paris (1987).
Steve McConnell, Programmation professionnelle, Microsoft Press, Les Ulis (1993).
M.-C. Gaudel, B. Marre, F. Schlienger, G. Bernot, Précis de génie logiciel, Masson, Paris (1996).
Henri Habrias, Introduction à la spécification 2, Masson, Paris (1993).
A.Cardon, C.Dabancourt, initiation à l'algorithmique objet, Eyrolles, Paris (2001)
L.Albert et al , cours et exrcices d'informatique en CAML, Vuibert, Paris (1998)
M.Quercia, algorithmique, cours et exercices en CAML, Vuibert, Paris (2002)
B.Warin, algorithmique, passeport ppour la programmation, ellipses, PAris (2002)
Th.Cormen et al, introduction à l'algorithmique en 2° cycle, Dunod, Paris (2002)

Chapitre 5 à 8

P. Lignelet, Structures de données avec Ada – Conception orientée objets, Masson, Paris (1990).
Jacques Ferber, Conception et programmation par objets, Hermes, Paris (1990).
Timothy Budd, Introduction à la programmation par objets, Addison Wesley, Paris (1992).
Masini, Napoli, Colnet, Léonard, Tombre, Les langages à objets, InterEditions, Paris (1989).
Grady Booch, Ingénierie du logiciel avec ADA, InterEditions, Paris (1988).
Grady Booch, Conception orientée objets et applications, Addison Wesley, Paris (1992).
M.-F. Barthet, Logiciels interactifs et ergonomie, Dunod, Paris (1988).
O. Foucaut, & al Conception des systèmes d'information et programmation événementielle, InterEditions, Paris (1996).
Gérard Weidenfeld et all, Techniques de base pour le multimédia, Masson, Paris (1997).
Jean-Pierre Vickoff, RAD – Développement Rapide d'Applications, Simon & Schuster Macmillan, Paris (1996).
R.L. Glass, R.A. Noiseux, Maintenance du logiciel, Masson, Paris (1983).
Bertrand Meyer, Conception et programmation par objets, EyrollesParis (2000).
M. Lai, Conception orientée objet : Pratique de la méthode HOOD, Dunod, Paris (1991).
M.Lai, UML la notation unifiée de modélisation objet, Dunod, Paris (2000).
M.Fowler, UML, Campus press, Paris (2001).
Morley,Hugues,Leblanc, UML pour l'analyse d'un système d'information, , Dunod, Paris (2002).
J.Gabillaud, SQL et algèbre relationnelle, ENI, Paris (2004)
C.Soutou , – de UML à SQL, Eyrolles, Paris (2002).
A. Burda, G. Färber, Delphi 2, Micro Application, Paris (1996).
Borland, guide du langage pascal objet, www.borland.fr, Paris (2002)
Dick Lantim, Delphi professionnel, Eyrolles, Paris (1997).
Dick Lantim, Delphi – programmation avancée, Eyrolles, Paris (1996).
Dick Lantim, Delphi3, Eyrolles, Paris (1998).
Gilles Betz, Delphi 1,2 et 3 – Edition développeur, Sybex, Paris (1997).
G. Deutsch, M. Gross, K. et M. Richter, Delphi 3, Micro Application, Paris (1997).

Borland, guide du développeur Delphi 7 sous Windows, www.borland.fr, Paris (2003)
Borland, guide du développeur Kylix 3 sous Linux, www.borland.fr, Paris (2003)
Th. Binzinger, Delphi 6 l'intro, Campus press, Paris (2002)
S.Teixera, X.Pacheco, Delphi 6 développement, Campus press, Paris (2002)
O.Dahan, P.Toht, Delphi 7 studio, Eyrolles, Paris (2003)
M.Martin, codes en stock Delphi 6, Campus press, Paris (2002)
M.Martin, le tout en poche Delphi 7, Campus press, Paris (2003)
G.Gardarin, Best of, Base de données, Eyrolles 5° éd., Paris (2003)